Project no. 034084
Project acronym: SELFMAN
Project title: *Self Management for Large-Scale Distributed Systems based on Structured Overlay Networks and Components*

European Sixth Framework Programme
Priority 2, Information Society Technologies

Deliverable reference number and title:	D.5.1 User Requirements
Due date of deliverable:	July 15, 2007
Actual submission date:	July 15, 2007
Start date of project:	June 1, 2006
Duration:	36 months
Organisation name of lead contractor for this deliverable:	FT
Revision:	0.5
Dissemination level:	CO

Contents

1 Executive summary

The purpose of the WP5 is to provide user requirements, demonstrators and evaluations in different use cases.

User requirements (D5.1) for each use cases actually cover the description of an applicative context, the proposition of some autonomic scenarios in this context and finally the requirements on components, transactions, overlay networks and self-* features associated to the implementation of the proposed scenarios in the considered applicative contexts.

Four use cases were considered during Task 5.1. Two of them are confirmed. The first one proposed by France Telecom concerns M2M systems. The second one proposed by ZIB concerns a distributed database system. Two additional use cases were investigated as replacements for the one that should have been proposed for the partner E-plus which left the Selfman project in its first year. The first one, proposed by the PeerTV company (contact established by KTH(P2)), concerns P2P video streaming (P2P TV). The second one, proposed by the Bull company (contact established by France Telecom R&D(P4)), concerns a J2EE application server. After investigation, the latter use case from Bull finally appeared not suitable for the Selfman project and was then discarded. The former use case by PeerTV is still under investigation.

The M2M use case was developed in collaboration with WP2 (components) and in connection with WP3 (transactions) and WP1 (overlay networks). Its goal is to develop large scale distributed M2M architecture. The illustrative M2M system, that is basically dedicated to the management of the thermal environment of buildings is responsible for transporting data from very numerous sensors to several M2M services which process data from sensors and send commands to the thermal equipments actuators. The M2M use case exhibits strong requirements towards autonomic (self-*) features and components ; and secondary requirements towards transactions and overlay networks.

The distributed database use case was developed in collaboration with WP3. Its goal is to develop a distributed transactional database which supports versioning. A wiki , which is the illustrative example, can then be added by a thin layer on top of it. The M2M use case exhibits strong requirements towards autonomic (self-*) features, transactions and overlay networks ; and secondary requirements towards components.

The P2PTV use case was developed in collaboration with WP1 and especially KTH. Its goal is to develop a P2P system for streaming of videos and TV. The P2PTV exhibits strong requirements towards overlay networks.

2 Contractors contributing to the Deliverable

France Telecom R&D(P4), ZIB(P5), KTH(P2)and PeerTV have contributed to this deliverable.

France Telecom R&D(P4) France Telecom has contributed on the definition of the M2M use case and associated requirements. Parts of this work has produced results that can be found in the appendices of this deliverable and associated to WP5 but that can also be considered as contributions to WP2 (on the Fractal component model and self-management architectural framework) and WP3 (self-optimization services).

France Telecom is leader of the WP5, editor of the Selfman wiki section devoted to WP5 and editor of this deliverable (T. Coupaye). France Telecom contributors to this deliverable are (in alphabetical order): O. Beyler (R&D Engineer), B. Dillenseger (Senior Researcher), T. Coupaye (Senior Researcher), A. Diaconescu (Junior Researcher), A. Harbaoui (PhD Student), N. Jayaprakash (PhD Student)), M. Kessis (PhD Student), A. Lefebvre (Senior Researcher), M. Leger (PhD Student), F.-G. Ottogalli (R&D Engineer).

There have been no deviations from the workplan but France Telecom consumed more human resources than expected on this task/deliverable. Indeed, the work on user requirements for M2M systems actually revealed itself more complex than forecasted. France Telecom does develop and exploit operationally M2M platforms and applications but it is worth mentioning that today's operational M2M generic platforms are in fact pretty basic with typically a few dozens to hundreds sensors sending data directly to one client service/application that consumes/processes this data. The M2M use case proposed in this Selfman deliverable explicitly tries to envision tomorrow's M2M systems that are expected to be of much higher size and complexity - and that then would required advanced features such as component-based architectures, overlay networks, transactions and autonomic (self-*) properties studied in Selfman. Such reflections required a greater effort than expected.

ZIB(P5) has contributed on the definition of the wiki use case and associated requirements. This work is based on the development in WP3 on transactions in structured overlay networks.

ZIB contributors to this deliverable are (in alphabetical order): M. Moser (PhD Student), S. Plantikow (PhD Student), T. Schütt (PhD Student).

There have been no deviations from the workplan, but due to E-Plus leaving, their input to the distributed database scenario is limited. Especially, in the areas of performance requirements and user profiles, real-world data would have been helpful.

PeerTV together with KTH(P2) has contributed on the definition of the P2PTV use case.

PeerTV contributors are: Andreas Dahlström, Johan Ljunberg, Sameh El-Ansary, Mohammed El-Beltagy together with Seif Haridi from KTH.

There have been no deviations from the workplan since PeerTV and the P2PTV use case were not included in the initial workplan. This use still being under consideration in the Selfman project as a replacement for the E-Plus leaving.

3 Results

This section specifies user requirements for each of the 3 use cases: M2M, Distributed Database and P2PTV. More precisely, each use case covers the description of an applicative context, the proposition of some autonomic scenarios in this context and finally the requirements on components, transactions, overlay networks and self-* features associated to the implementation of the proposed scenarios in the considered applicative contexts.

3.1 M2M Use Case

This section introduces the Machine-To-Machine (M2M) use case proposed by France Telecom. The use case is made of an applicative context, some autonomic scenarios in this context and finally the requirements in terms of i) functional and non functional desired properties of a M2M system and ii) the required mechanisms to support the proposed scenarios. The latter part is to be taken as requirement to the other Selfman Working Packages and especially as requirement on components (WP2), autonomics features (WP4), transactions (WP2) and overlay networks (WP1).

3.1.1 Applicative Context

The applicative context we consider is that of a large M2M (Machine-To-Machine) system dedicated (essentially but not only) to the management of the thermal environment of buildings (homes). The big picture of the considered M2M system is given in Figure 1.

The M2M system is responsible for transporting data from sensors to several M2M services which in turn process data and send commands to the thermal equipments. At the edges of the system, thousands (or millions) of buidlings are equiped with sensors such as thermometers and smoke detectors ; and actuators on thermal equipments such as heaters and boilers. All these data are presents in a private local area network (called the *domestic environment* in the sequel). A gateway is present in this environment which role is to export these data to interested M2M services, provided by third party service providers, that will use process/use these data. Gateways ('GW' on Figure 1) can be seen as peripheral nodes of the M2M system. Three M2M services are considered in the use case. A fire alarm service detects fires by means of in-houses smoke detectors and possibly correlations with the thermal regulation service. A weather forecast service provides local, regional and national data about current (observed) and forecast weather. A thermal regulation service uses data from in-houses sensors (e.g. thermometers) and possibly weather forecasts to provide a thermal regulation of the domestic environments by sending commands to in-houses thermal equipments (actuators on boilers, radiators, etc.).

Different actors appear in the M2M use case:

- individuals (end-users) who want their domestic thermal environment to be managed (including fire detection). They of course agree to have their homes equipped with sensors (e.g. thermometers, smoke detectors) and actuators on their thermal equipments (e.g. boilers, radiators). They also agree to have their sensors data exported into the M2M infrastructure so as to be possibly used by third party service providers.
- M2M service providers: thermal regulation, fire alarm and weather forecast.
- the infrastructure operator who is in charge of operating the system, i.e. of deploying and managing the infrastructure so as to guarantee its correct behaviour (including QoS).

The M2M system architecture is typically a data-flow (Pipe & Filter) architecture made of interconnected nodes that receive, process and send data. The

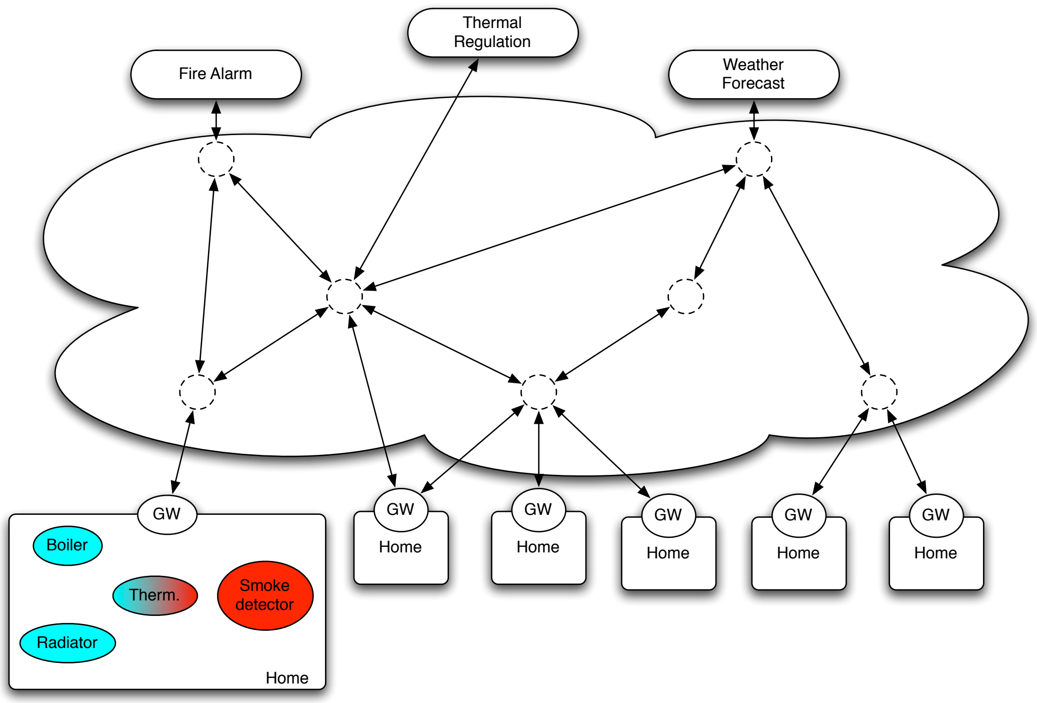

Figure 1: Big picture of the M2M use case.

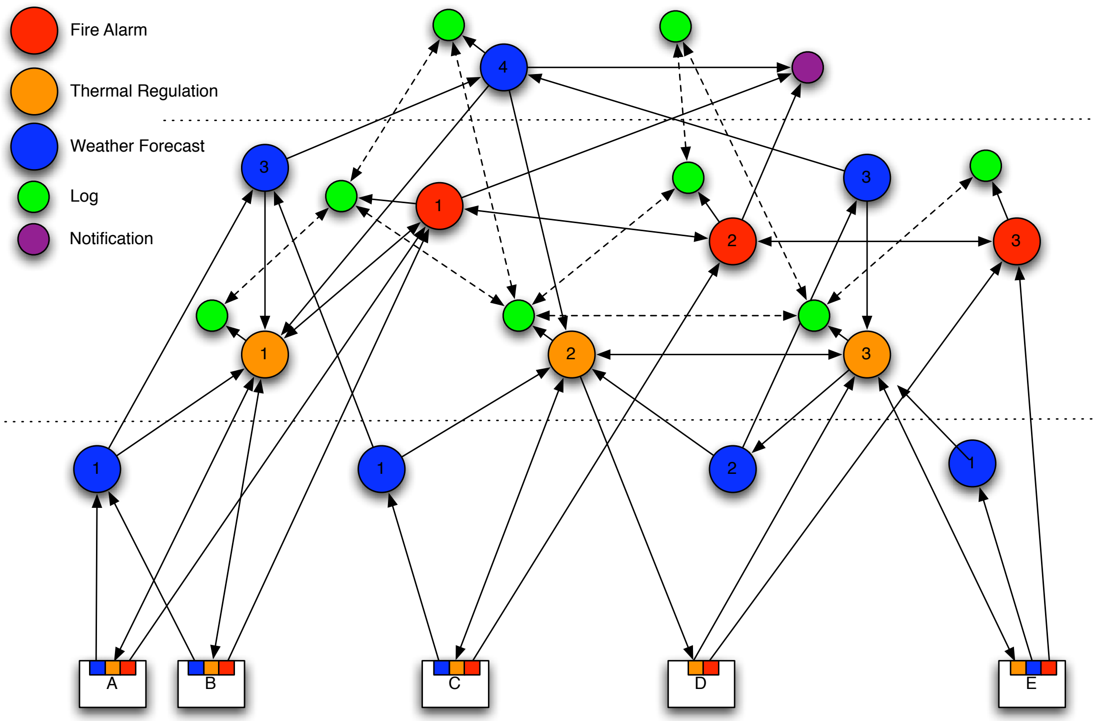

Figure 2: Detailed generic architecture of the M2M use case.

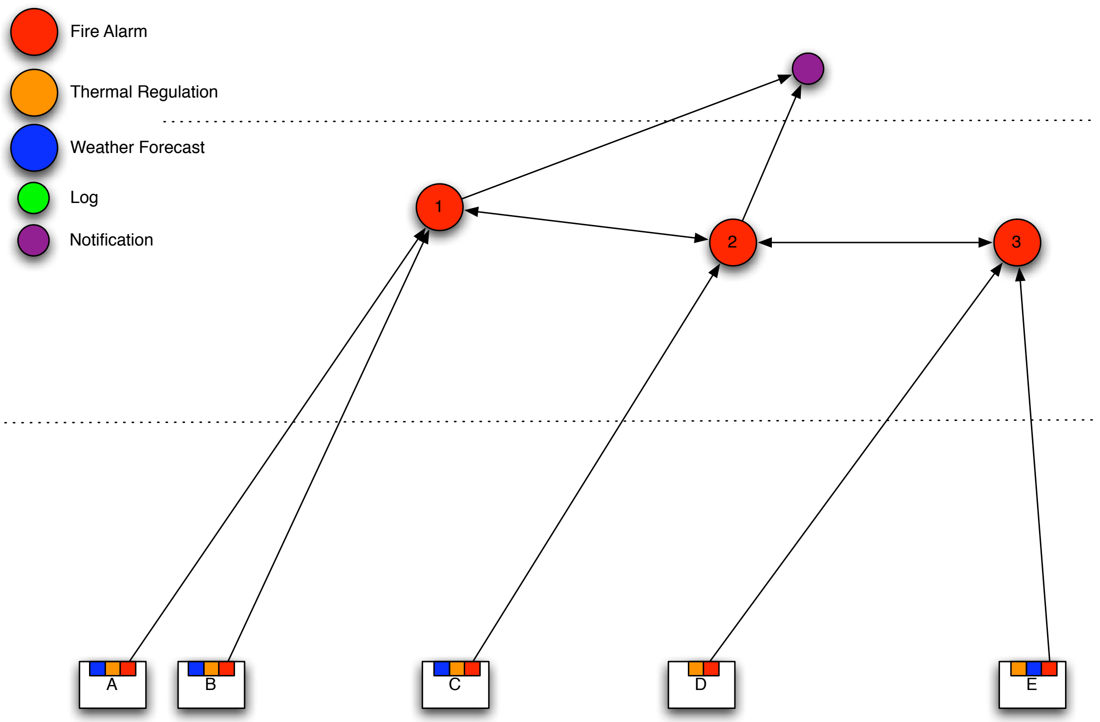

Figure 3: Fire Alarm service architecture.

complete system (see Figure 2) involves 5 interacting services: 3 are applicative M2M services: *thermal regulation*, *fire alarm*, *weather forescast*, and 2 infrastructure (technical, generic) services (or *enablers*): *logging* and *notification*. The services are distributed: each service is implemented of a set of interconnected distributed nodes. The complete M2M system is made of the interconnected service nodes.

An important feature is that different criticities are associated to services (and hence possibly to the data they manipulate): the fire alarm service is of higher priority than the thermal regulation service which is itself of higher priority than weather forecast service. Also, data emitted from the domestic environment could be categorized as data devoted to a unique service versus data shared among services. For instance, temperature data are used by thermal regulation, fire alarm and weather forecast ; while smoke detection data is used only by fire alarm. Due to the different criticities and business decisions between service providers to share or not data, M2M infrastructure elements (service nodes) may be shared or not between services. Another feature, due to the large scale of the system, is the hierarchical data diffusion pattern used - with typically 3 layers: local, regional, national (separated by dotted lines in Figure 2).

Fire alarm The purpose of the fire alarm service is to detect fire situations in domestic environments and to notify these alarms to third parties, typically the appropriate fire departments.

Fire detection is based either on smoke detectors only (by default), or by correlating data from smoke detectors and temperature data from thermal regulation.

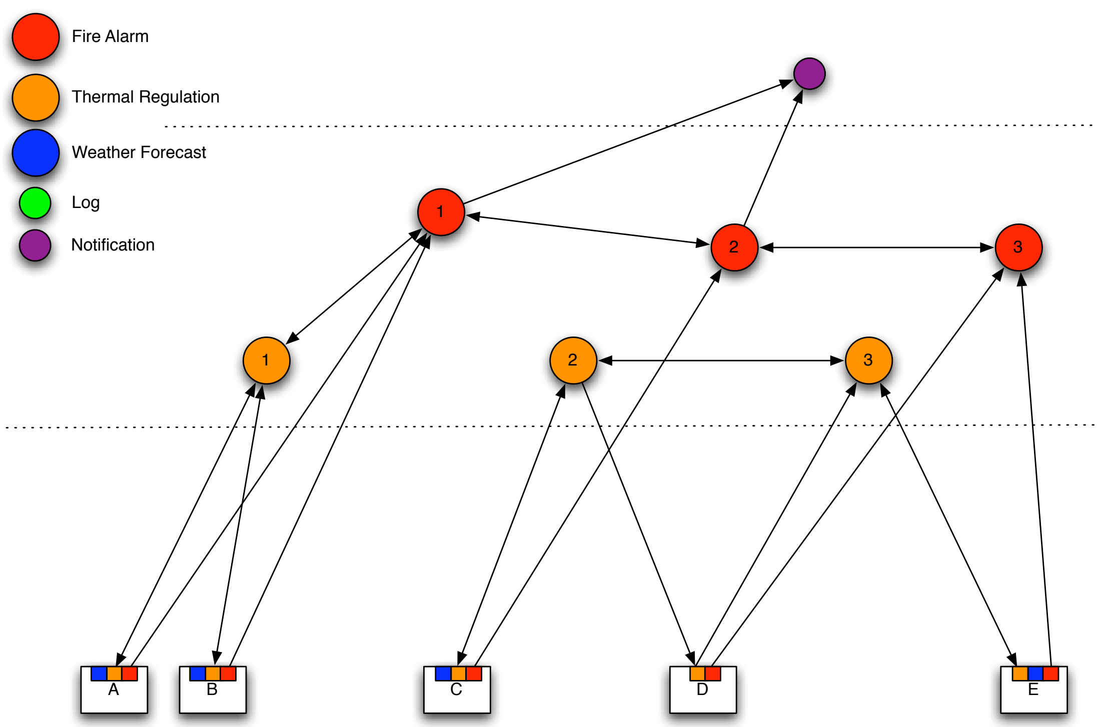

Figure 4: Fire alarm use case.

Indeed, to prevent false alarm detection, thermal regulation can help in discriminating situations (see Figure 4). Smoke can be detected in a house because of overcooked food in the oven (!), but without significant temperature increase. This situation could probably be classified as a false fire alarm detection. In the other hand, if the thermal regulation service provides in temperature historical representing a sharp increase, this probably describe a true fire alarm detection even without smoke detection. In a basic way, fire detection is based on smoke detectors. When smoke is detected in a house, a fire alarm node (red node in Figures 3 and 4) receives an alarm message from it and then a notification is sent. This situation is depicted by red node 2 in Figure 3. Node 3 is not able to notify by it self. It has to delegate the notification scenario to an alarm node linked to the notification service (nodes 1 or 2 in this example).

As the fire alarm service is critical and prioritary, data processed by fire alarm service nodes are of high priority. Moreover, in overload situations of the fire alarm service, resources can be 'stolen' from other services. In order to (try to) prevent such situations, self-optimization mechanisms such as intra-service routing or load-balancing can take place. Intra-service routing depicts situations in which an overloaded node would send the data it receives to a pair node belonging to the same service, e.g. the fire alarm service in this case. Finally, a specific network of nodes is dedicated to the fire alarm service (cf. Figure 3) with specific resources dedicated to it. A constraint on the fire alarm service which is critical us that the sub-graph of the complete system graph corresponding to the fire alarm shoud always be connected.

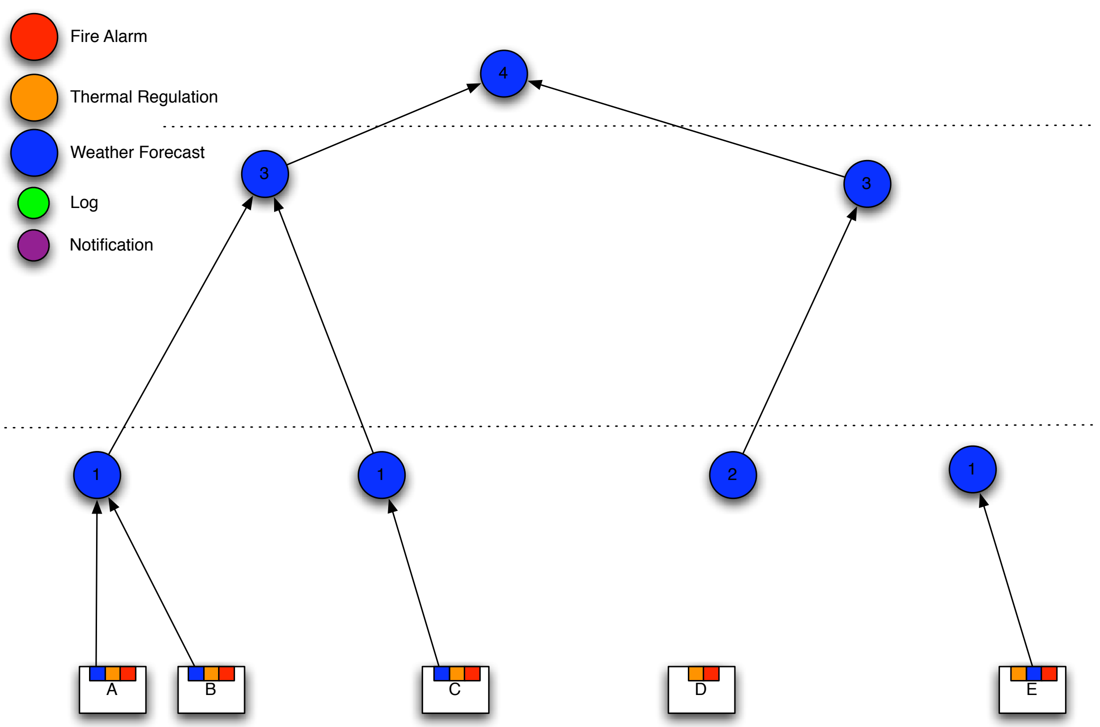

Figure 5: Weather forecast architecture.

Weather forecast The weather forecast service provides data about the current weather and future tendencies. The forecasts are build upon actually observed data provided by local sensors (thermometers, barometers) and possibly satellical data (cf. Figure 5). Local data are more accurate but with a narrow range of validity. Forecasts based on it are for short periods of time (typically couple of hours). Regional data aggregate local data. It help to estimate longer period forecasts (typically about one day). The national level uses both regional data and satellites data. Forecasts can be made on longer periods such as several days.

The weather service is organized hierarchically to match the accuracy of the weather forecasts discussed above. In each level (local, regional, national), nodes are functionally equivalents, i.e. they implement the same function/algorithm

A strong connectivity hypothesis exists between the regional and national levels. No hypothesis exist between local and regional levels. A node can be isolated and so need nodes from other services to keep the connectivity. This is QoS and pricing concerns.

Weather forecast can be used in several ways. The most simple one is to provide weather forecasts to customers (subscribers). In that case, a weather forecast node have to notify the forecast to a customer thanks to a notification service (e.g. blue node 4 in Figure 6). Another way is to provide weather forecast to the thermal regulation service: thermal regulation is essentially based on the actual observed local temperatures, but it can include weather forecasts so as to anticipate evolutions of the weather. The accuracy and the time validity of the forecasts depend on which node is providing the data. On the Figure 6, nodes 1 and 2 provide local data with

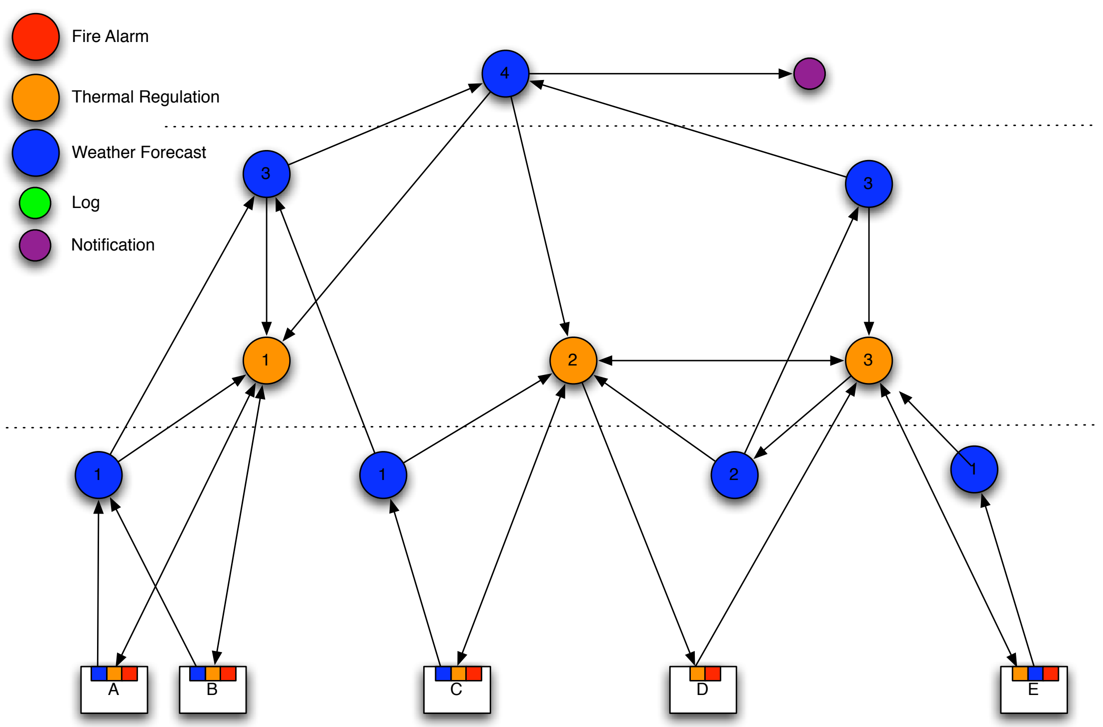

Figure 6: Weather forecast use case.

a short range of validity. Nodes 3 give accurate forecasts about a region with a middle range of validity. Node 4 gives national level forecast with a longer range.

Thermal regulation Thermal regulation service providers (typically thermal equipment vendors) are able to manage domestic equipments to achieve thermal regulation specified by policies. Policies are based on goals to be reached such as maximum amounts or acceptable ranges of power consumption. To make regulation decisions, data are collected from domestic environments - and also possibly from the weather forecast service. As different levels of weather forecasts exist in the system (regional, national), the thermal regulation service can subscribe to the one(s) that is(are) relevant.

Thermal regulation exhibits an autonomic behaviour since it implies sending commands to thermal equipments as a reaction to changes in observed thermal conditions. Commands are functions of the delta of in-house temperatures between the observed temperatures and the desired ones. Commands are specific to the target equipements. A thermal regulation service provider may or may not be able to build the proper command to send. Both situations are depicted in Figure 8. Orange node 1 represents a node able to receive data from a house and to send commands to it. Orange node 3 is able to receive data and send commands to house E but not to house D. Commands to house D have to be send through orange node 2. In this case, orange node 3 and 2 have to collaborate (i.e. node 3 will build and send comands to house D on behalf of node 2).

Finally, in thermal regulation phases, the way the energy will be consumed has

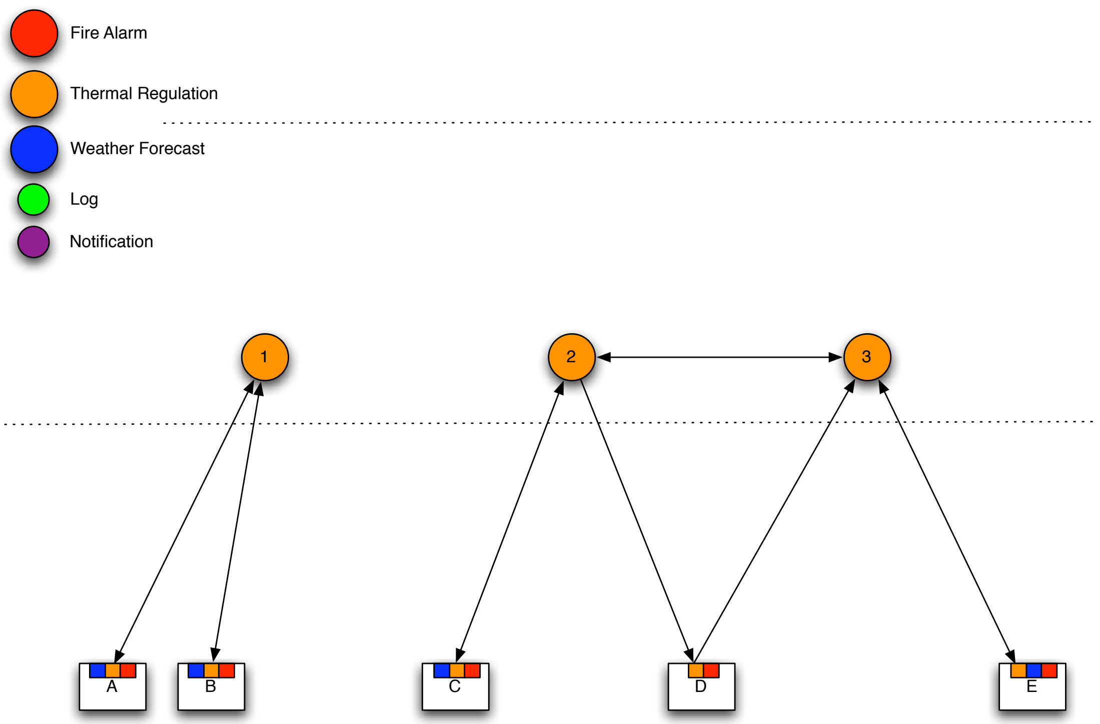

Figure 7: Thermal regulation service architecture.

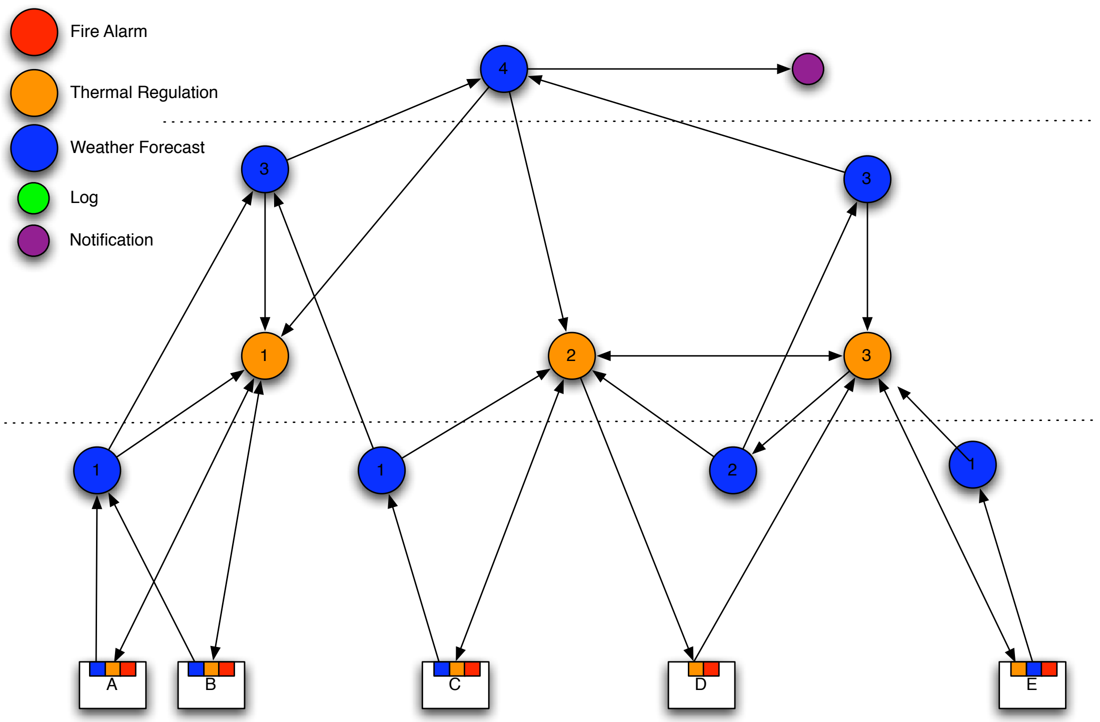

Figure 8: Thermal regulation use case.

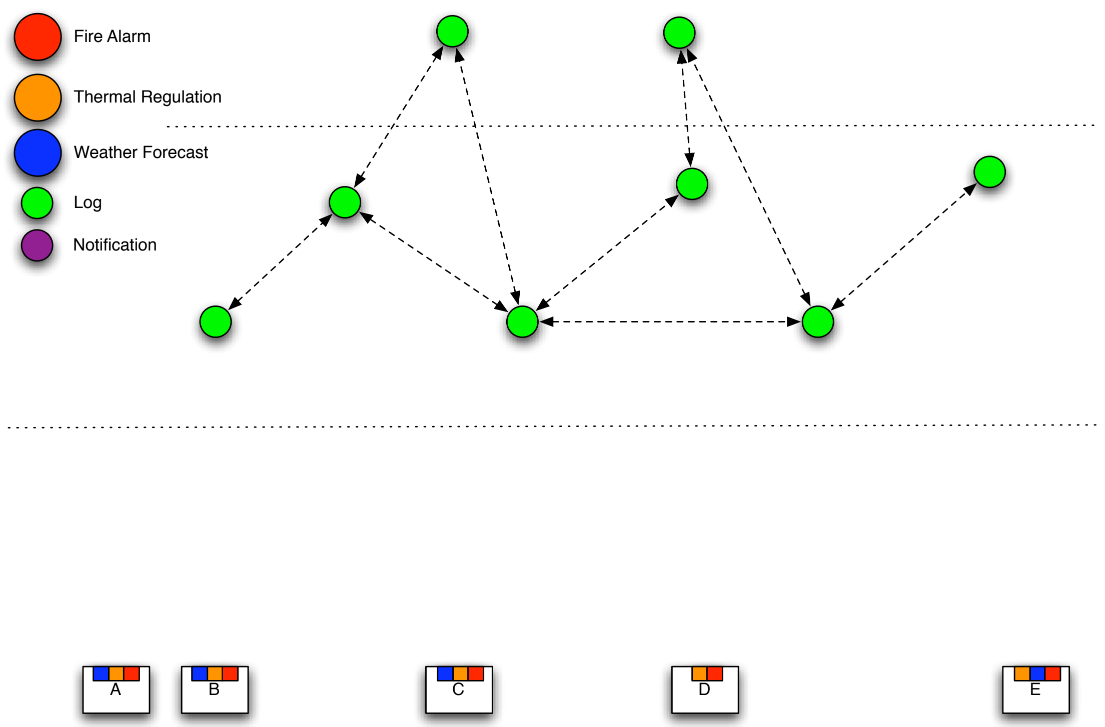

Figure 9: Logging service architecture.

to be managed to avoid to overload the energy distribution network. Thus, thermal regulation services (from different providers) have to collaborate, as much as possible, to smooth the energetic demands from the houses. This involves collaboration between thermal regulation service providers to plan the commands to send, and with the energy providers to advertise the energetic requirements.

Thermal regulation use data from multiple sources: from sensors (e.g. temperature), from the weather forecast service, from the end-users (e.g. desired thermal ambiance) and from thermal equipments (e.g. power level). Data produced by thermal equipments are pushed at agiven frequency. When a thermal regulation action is engaged, a timeout is set to prevent to send new commands before the effects of the first have an effect (hysteresis). Data from houses, as well as commands, have a time-to-live to avoid unrelevant data or commands.

Logging service The logging service is a classical, generic and common facility share by all M2M services that provide for interfaces used to manage (creation, destruction) and actually use (read and write queries) logs (i.e. traces of services behaviour).

In our M2M use case, the logging service could be implemented as an (P2P) overlay network[1]. When a M2M service node (or client node) needs to access to the logging service, it just has to find a reference to one logging node and then to bind to it. Since all nodes of the logging service are functionally equivalent, a client

[1]This possibility will be further investigated according interest from other Selfman Working Packages.

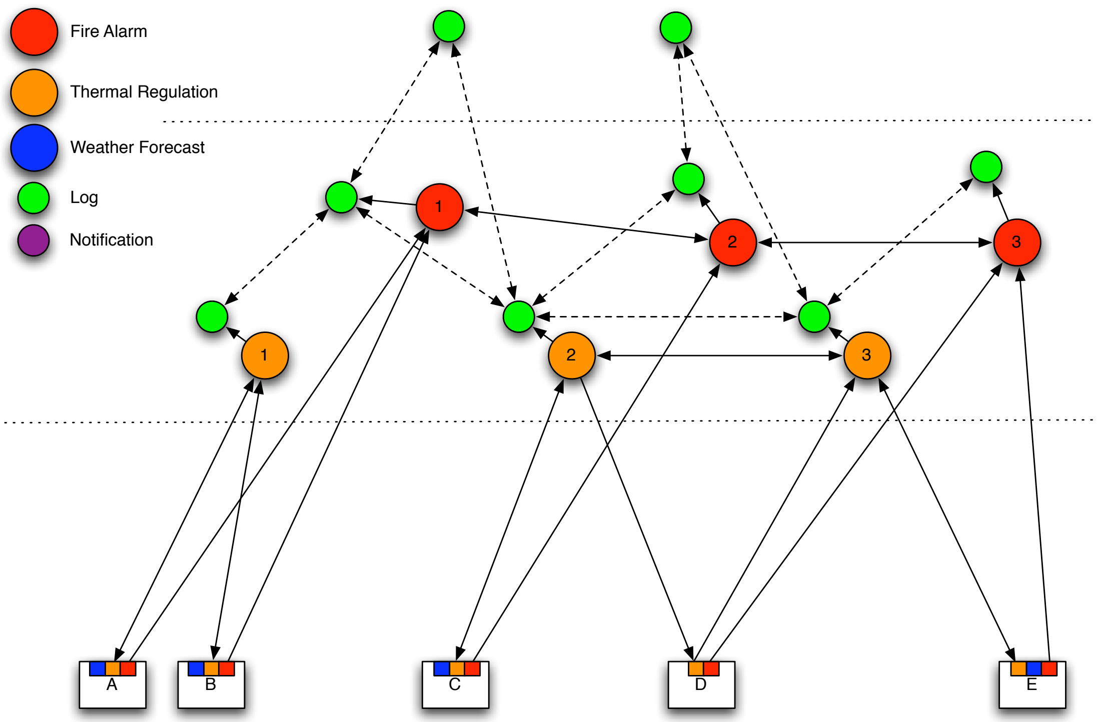

Figure 10: Logging service architecture.

node can link to any logging node. Log queries are handled by the overlay, i.e. a query can be adressed to any logging node, queries and query results are routed by the overlay.

Since logging data can have different priority levels, a QoS level associated to each link between an M2M service node and a logging node. A specific link has then to be established for each level of log needed by the node.

A transactional logging service could also be envisionned with concurrent queries and log persistence[2].

Notification service The notification service is generic, common technical service (enabler) that can notify messages to subscribers on different media: SMS, MMS, e-mail, fax, text-to-speak etc.

As the logging service, the notification service could be architectured in as an overlay[3].

3.1.2 Autonomic Scenarios

An M2M system, such as presented in the previous section, typically consists of a distributed set of data processing nodes linked together in an arbitrary graph/network.

[2]This possibility will be further investigated according interest from other Selfman Working Packages.

[3]This possibility will be further investigated according interest from other Selfman Working Packages.

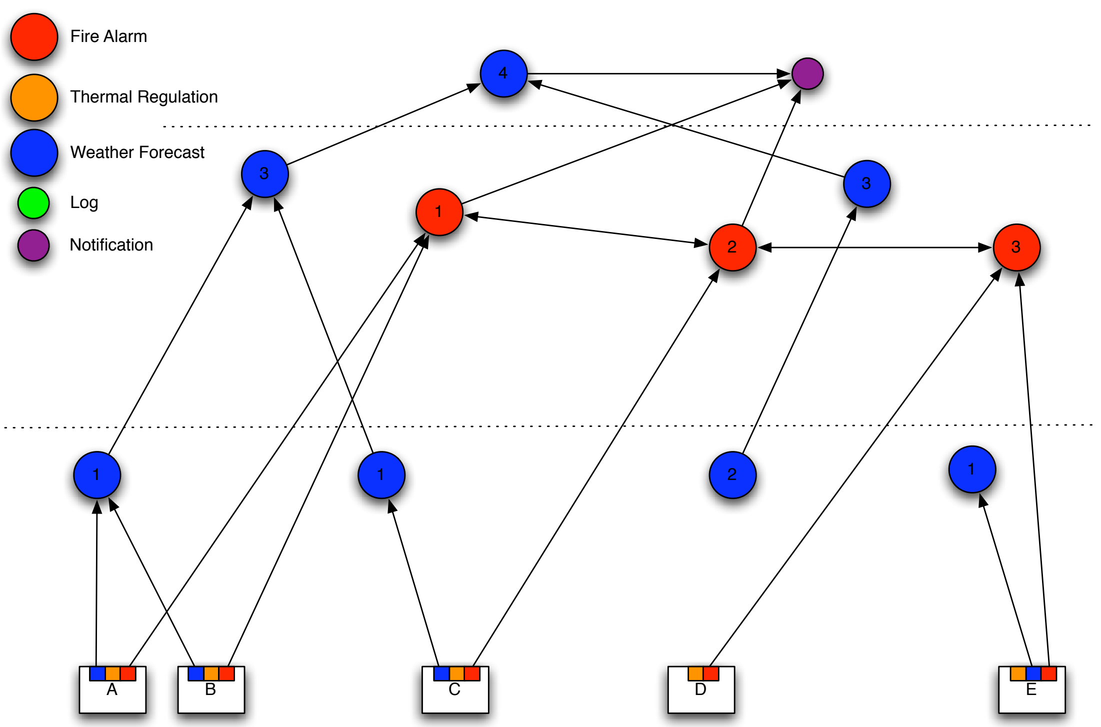

Figure 11: Notification service architecture.

Each node consumes messages that contain data and that come from an arbitrary number of incoming links, then process data in a so-called *proceed* operation implementing some business process, and then produce an arbitrary number of messages containing data on a number of outgoing links.

Such a distributed infrastructure is a typical target for autonomic computing, especially in our M2M use case where the nodes are really numerous (possibly millions of homes), heterogeneous in terms of functionality and criticality, and widely distributed on an arbitrarily complex network topology. An autonomic M2M infrastructure may dynamically and autonomously evolve by means of reconfigurations (adding, removing, updating of nodes and links) in order to adapt to changing execution conditions. Here come commonly identified autonomic features, classified according to the common principles of self-configuration, self-protection, self-healing (or self-repair), and self-optimization.

Self-protection scenarios

Fault prevention through M2M nodes reinstantiation This scenario relies on detecting special conditions that are likely to evolve towards a node failure. For instance, if the available memory on the computer hosting an M2M node is going lower and lower, possibly because of a memory leak in the node implementation, a new node may be instantiated on another host computer in order to replace the node that is expected to fail.

On the sensor part, this scenario requires fault-prevention detectors, e.g. detectors that anticipate over a possible failure. The memory usage example given above may be transposed to all typical computing resources, such as CPU usage, number of processes, etc. On the actuator part, this scenario will reconfigure a sub-part of the network topology in order to replace the suspicious node with the new one. This replacement is achieved through a two-steps process consisting in first instantiating the new node and then discarding the local node. The two steps shall be performed in a sort of a transaction so that (1) at no moment both nodes are active and actually processing incoming messages, and (2) no message is lost. It may require also a state transfer between nodes in case of stateful proceed functions.

In our use case, this scenario may be applied to the notification service. The service must always be on and quickly responding. So, the autonomic control must anticipate on possible failures.

Overload prevention through graceful service degradation An M2M infrastructure typically hosts several services that are likely to share a number of computing and networking resources. When a serious overload occurs in the M2M infrastructure, a possible solution to prevent a global crash of the infrastructure and hosted services is to apply priority policies between services. Low priority services may be stopped or disconnected in order to reaffect computing and networking resources to the most critical services. This is a self-protection mechanism since an overload condition generally results in performance degradation that in turn amplifies the overload condition, and so on, until partial or general failure of the system. Here, the autonomic control ensures most critical services still work. This strong enforcement may be applied for severe overload only. Light overload may be overcome by self-optimization policies (see section 3.1.2).

A variety of overload indicators may be used in this scenario: bandwidth usage on network equipments or M2M nodes' hosts, available memory in JVMs or hosts, CPU usage on hosts, etc. Beyond these low level indicators, M2M-level indicators may also be used such as the number of pending messages in the in-buffer, or the processing time of messages. Regarding reliability, since an M2M-based application is modeled as a DAG, it could be interesting to guarantee that between several semantically redundant paths in the graph, one is always active i.e., with no stopped or broken nodes.

In our use case, a progressive degradation could be made, starting by disconnecting the weather forecast service, then the thermal regulation service, to keep all resources for the fire alarm service working. Not only some communication links can be disabled in the M2M infrastructure to save network resources, but nodes' hosts may be affected to the fire alarm service.

Self-healing scenario

Node Failure Recovery As a complement of scenario 3.1.2, this scenario consists in reacting when a node has unexpectedly failed. It simply consists in instantiating a replacement node. The consequence may be different depending on whether the service is supported by a single node or by a set of nodes. In the former case, there will be a service disruption, while in the latter case, the service may be maintained, possibly but not necessarily disturbed by a temporary load increase.

Common failure detectors are the hello protocol (are you alive?) and the heartbeat protocol (I'm alive). Such protocols may be implemented are various levels (e.g. through ICMP or IP network datagrams, JVM monitoring, or via dedicated messages between M2M nodes). The higher level, the more reliable the protocol is, because it means all lower levels are OK while you don't know about upper levels. For instance, the network connectivity may be correct while the JVM running the M2M node has crashed.

In our use case, this scenario may be applied to the fire alarm service. The service must always be on and quickly responding. So, the autonomic control must ensure that the set of nodes supporting the service is always sufficient. The fire alarm service may temporarily afford a node failure among the set of nodes, while a new node is being reinstantiated.

Self-optimization scenarios

Drop or delay data processing according to QoS A QoS specification may be taken into account in overload situations, so that some messages can be delayed or dropped when a congestion situation occurs. In some M2M-based applications, some messages may be more important than others or, more precisely, messages require a variety of delivery constraints. For instance, temperature measures may be dropped from time to time without noticeable impact, while a heat excess alarm (fire threat) shall be neither lost nor delayed. However, an instant measure such as temperature should be either dropped or delivered in a timely fashion. Another example is a meter reading for electricity or gas consumption that must not be lost but may be delayed for a couple of days. So, we see from those examples three kinds of Quality of Service on delivery reliability and delay: (1)as soon as possible, (2) now or never, (3) whenever possible.

Implementing such a scenario is likely to involve filtering nodes enforcing QoS policies accordingly to the congestion level, in order to provide a generic solution for the required QoS parameters. A scheduling policy with dynamic priorities between nodes can be a solution to manage processing of data according to the execution context.

In our use case, this scenario could be typically applied to the transmission of local weather measurements (temperatures, pressure, humidity). With less data, the weather service and the thermal regulation service can still work, possibly with lower accuracy, while still keeping the fire alarm service work. Of course, data coming from fire detectors shall not be concerned by this message drop or delay feature.

Increasing processing power through node clustering and load balancing This scenario consists in instantiation new M2M nodes implementing a given proceed method P in case the (set of) existing node(s) implementing this method is/are overloaded. This overload condition may be detected when an incoming message buffer is full. New nodes shall be instantiated on a distinct physical host in order to actually increase the amount of computing resources.

This reconfiguration introduces redundancy for the identified bottleneck P method, which requires a routing feature to be inserted (when not already present), typically through a message routing node performing some load balancing of incoming messages between the redundant nodes. This scenario subsequently implies finding available target hosts for deployment of new nodes and accordingly updating links. The modification involves not only the links topology, but also the nature of some links, since some local shared memory-based links may become network-supported remote links (TCP socket, JMS topic or queue).

It has to be underlined that such a reconfiguration is neither always possible nor always relevant. For instance, replicating a database or an SMS-sending node might be either practically not feasible or not relevant because the actual bottleneck resource cannot be replicated (e.g. hardware or location-dependent resource). The autonomic decision process will also have to take into account that the resulting wider distribution of the M2M infrastructure also possibly results in a greater usage of communication-related resources, greater communication latency and lower communication bandwidth.

Optimize computing resources usage and communication through co-localization When several nodes are executed on physically distributed hosts and are linked to one another to exchange messages, it may be smart to change the configuration, when possible, in order to co-locate them and optimize communication. This may also enable saving power by switching unused hosts off. The links topology will be updated and their nature possibly changed for the sake of efficiency. Some network-supported links would typically be changed into shared memory-based links. The autonomic decision process will have to check that all necessary resources are available at the target host, and that no compatibility or physical attachment constraint forbids the reconfiguration plan.

Self-configuration scenario

A self-configured logging overlay network M2M nodes may be used to implement a service based on a peer-to-peer network model. Here, distributed M2M nodes all provide the same service and cooperate in order to support the service. As with classical P2P networks, the expected benefits may be efficiency and robustness thanks to a smart service distribution between peers. Pushing the autonomic approach there could consist in having a self-configured network of peer M2M nodes to provide a service to fulfill specific non functional objectives.

In our use case, we may apply this P2P approach to the log service. This service provides a number of other services (weather forecast, thermal regulation, fire alarm) with a persistence facility for events that may be useful to consult some time later (e.g. for troubleshooting purpose). As a consequence, this service must be efficient, reliable and widely available in the global M2M infrastructure. Peer log nodes may be linked with each other in an arbitrary, self-configured topology.

3.1.3 Requirements

This section identifies the requirements associated the autonomic scenarios in the considered M2M applicative context presented in the previous sections. The section is organized in two sub-sections. This first one identifies requirements in terms of functional and non functional desired properties. The second one then lists some mechanisms/functions required to support the desired properties. This second sub-section exhibits requirements for the other Selfman Working Packages. Hence, it is organized in 4 paragraphs: requirements on component programming, transactions, overlay networks and autonomics (self-* mechanisms).

Required Properties

Flexibility, modularity, manageability, maintainability The desired properties, that we may call *architectural properties*, refer to the ability to manage (in the most general meaning) distributed software configurations. *Flexibility* and *modularity* refers to the ability to configure and dynamically reconfigure homogeneously both individual components and component assemblies forming an M2M infrastructure. In order to enhance an M2M infrastructure with autonomic features, it should be possible to easily manipulate complex software configurations in terms of components and connections between components (i.e. M2M nodes and links). Since autonomic features required runtime adaptations, configurations and dynamic reconfigurations should both be supported. *Manageability* refers more generally to software deployment where the term *deployment* covers all the activities that have to be carried out after the development itself e.g. packaging, delivery, installation, configuration, activation, update and upgrades (i.e. reconfigurations), deactivation, uninstallation. *Maintainability* refers to the ability of an M2M infrastructure to be easily repaired and evolved. These features are of course of primary importance in autonomic settings.

Reliability, integrity This desired property essentially refers here to the ability to guarantee a *correct behaviour* of a system in a given environment. The reliability property is assessed against its *integrity* supposedly defined by *integrity constraints* on the state and behaviour of a system. In our M2M context, some integrity constraints, such as the ones that concern reliable data routing and processing (e.g. 'no data must be lost', 'all data must be processed'), may be considered as functional. Some others may be considered as non functional: typically the ones that concern the reliable reconfigurations of an M2M infrastructure.

Availability This desired property refers to the ability of a system to ensure its function 'the maximum possible amount of time'. Availability is generally seen as a ratio between *Mean time between failure* (MTBF) and *Mean time to repair*. M2M systems we consider put serious requirements of the availability of the supporting M2M infrastructure especially in presence of different quality of service (QoS) on data routing and processing, and priorities between M2M services.

Scalability This desired property refers to the ability to either handle growing amounts of work, in a graceful manner, or to be readily enlarged. In our M2M context, this feature refers i) to the ability to support the equipments of thousands to millions of domestic environments and ii) to support the addition of new M2M services (or applications e.g. Thermal regulation, fire alarming, weather forecast, logging, notification in our applicative context). Both generally in a growing number of M2M nodes and links. This requirement is connected to the requirement on manageability and accountability since it puts heavy requirements on deployment and monitoring capabilities. It is also connected to the reliability and availability requirement for QoS requirements on data processing and management of priorities between M2M services have to be supported even with large and growing number of M2M nodes.

Observability, accountability As an M2M possibly involve, as in our sample applicative context, multiple actors (people living in the houses, people from the fire department or weather forecast service, etc.) including actors that support critical businesses (e.g. fire alarming), it is of paramount importance for an M2M infrastructure i) to provide means for easily add,remove, manipulate probes and sensors so as to be able to define responsibilities in case of malfunctions (e.g. data loss) and failures (e.g.crash of nodes or links) and ii) to assess resource consumption per services in a billing perspective ('pay-per-use').

Autonomicity This desired property refers to *self-* properties*, i.e. the ability for a system i) to observe its structure, behaviour and environment, and ii) to take corrective actions if needed. Autonomicity is of primary importance in large scale, dynamic, open M2M systems, such as the ones envisioned here, in which a "manual" (by human operators) configuration and management is almost impossible. Autonomic computing seeks fundamentally to automate as much as possible the deployment and management of software systems so as to lessen human interventions and afferent costs - which is very relevant in M2M settings.

Required Mechanisms

Components The M2M use case exhibits strong architectural requirements in terms of flexibility, manageability, scalability, accountability. These call in turn for a sound architectural framework which allows for M2M system configuration, deployment and management. Component models look like the ideal candidates to support these requirements - especially:

- reflexive components models w.r.t. observability and dynamic reconfiguration,
- hierarchical (or recursive) component models w.r.t. scalability, i.e. the uniform management of large scale distributed systems at arbitrary levels of abstraction - typically through the concept of *management domains* (possibly overlapping),

- open/extensible component models w.r.t the great variability in terms of components types, components life cycle, programming languages, etc. that has to be supported in large scale autonomic distributed systems.

Advanced component models generally comes with advanced languages and tools such as Architecture Description Languages (ADLs), packaging and deployment models and management frameworks, powerful monitoring and dynamic reconfiguration support, QoS contracts support, etc. which are very valuable and (arguably) required mechanisms in autonomic systems.

In the M2M use case, all scenarios require configuration, deployment and management capabilities as provided by components as a basis to almost all desired properties listed in the previous paragraph e.g. configurability and manageability through reflexive components, scalability trough hierarchical components, reliability and availability through the intrinsic isolation provided by component-based architectures.

Autonomics For a system to be autonomic (or self-adaptable or featuring self-* properties):

- it must be adaptable, i.e. its construction has to be based on a explicit structural (architectural) model that exhibits its sub-elements that can be subjects to adaptation and operative mechanisms that allow for the realization of these adaptations ;

- it must have this knowledge of itself (e.g. through reflexive mechanisms) and of its environment in order to detect changes in its environment or in its own behaviour so as to take corrective actions - essentially expressed as reconfigurations.

We consider here that an autonomic system is composed of an *autonomic infrastructure* superimposed on a *target (component-based) system*. The autonomic infrastructure is responsible for implementing a *control loop*, i.e. instrumenting the components of the target system for monitoring, detecting and notifying events, diagnosing the system based on these events, and making decisions to determine what and how corrective actions need to be executed, and finally executing the corrective actions on the components of the target system.

An autonomic control loop conceptually should allow for advanced observation, diagnosis, decision making and reconfiguration (not to mention event and actions transport mechanisms we do not detail here).

By *advanced observation*, we mean powerful monitoring capabilities where individual probes/sensors can easily be programmed, deployed and configured (e.g. push/pull, change polling frequency, etc.). Also, diagnosis capability required in autonomics setting does generally not boil down to the simple observation of individual probes but requires probe aggregation and event correlation mechanisms (this is the case for instance in the M2M autonomics scenarios introduced in the previous section).

By *Decision making*, we mean high level, typically declarative, formalisms/languages that allow for the specification and execution of *heuristics* and *policies*. Although complex mechanisms coming from artificial intelligence such as neural networks, bayesian networks, etc. can be used ; we consider simpler mechanisms such as Policy languages (e.g. Ponder or PDL), *deductive rules* (e.g. JBoss Rules) or *active rules* (or Event-Condition-Action rules) are good candidates to implement the core reactive behaviour of an autonomic control loop.

Moreover, a *simulation mechanism* would allow for checking that a given reconfiguration decision would be actually correct and fruitful. As for typical artificial intelligence techniques, the idea is to generate possible action plans and to evaluate them, using modeling and simulation in our context. Our M2M use case particularly suits queuing network-based modeling and simulation. For modeling purpose, we need to characterize the performance and resource usage of M2M nodes. Since M2M nodes embed arbitrary business code with no assumption about its performance and resource usage, we will adopt either (1) an online characterization approach with a real workload and probes for observation purpose, or, (2) even better and whenever possible, an offline characterization approach with a self-regulated traffic generator sending messages to an M2M node, still including probes for observation.

As discussed in the previous section and examplified by the proposed M2M autonomic scenarios, autonomicity is a very valuable feature in the M2M use case. Note that all the mechanisms we mention here, again, need to be flexible and in particular dynamically reconfigurable: in the long run, we might probably consider that an autonomic control loop has to be itself autonomic.

Transactions The M2M use case exhibits strong requirements in terms of reliability and integrity. When looking deeper at these requirements, we discover they match pretty well the *transactional properties* (or so-called ACID properties). The M2M use case exhibits (at least) 5 use cases for transactions:

1. *Transactional reconfiguration*: this is a classic requirement towards ACID transactions for concurrency and recovery purposes. Reconfiguration transactions, which are sequences of reconfiguration actions (e.g. add, remove,replace components and bindings between components), have to be atomic, consistent, isolated and durable. Reconfiguration transactions can run concurrently so reconfiguration transactions have to be isolated. Consistency in this context is based on integrity constraints that can be generic (typing constraints, there cannot be cycles in the graph of components hierarchical containment) or application specific. Durability is based on persistent logging used for recovery. The need for transactional reconfiguration appears in all autonomic scenarios of the M2M use case.

2. *Inter-nodes transactional data routing and processing*: An M2M system is made of a set of nodes organized in an almost arbitrary graphs. A transactional data processing behaviour is sometimes required. In our M2M use case, the processing of fire alarm data from the sensors (smoke detectors) to the fire department through the M2M network of nodes should be transactional. There might be room for non conventional transaction models here. For instance compensating transactions (as in SAGAs) is often preferable to classic flat transactions in an M2M context. There might be also room for nested transactions.

3. *Intra-node transactional data processing*: An M2M node is typically made of 3 sub-components (in turn each made of sub-components): one that handles data reception (possibly from multiple connections), one that actually process the data, and one that handles data emission. Du to the no data loss requirement, a transactional behaviour (especially atomicity) is required between these 3 sub-components inside a node component.

4. *Transactional logging*: Our M2M use case specifies a single dedicated network of M2M nodes for logging. The logging service, which receives read and write logging accesses, could be transactional, essentially in order to support concurrency. In one of the proposed autonomic scenarios, the logging service could be self-organized as a overlay network (see below) which leads to transactions in overlay networks.

5. *Transactional deployment*: As we have seen before, manageability and especially remote deployment is a strong requirement in M2M environments. In our M2M use case, a transactional deployment (initial installation or upgrades) of software inside domestics environments would be a valuable asset. The main requirement concerns atomicity: software in all (or at least large sets of homes) should be upgraded atomically or none. As logging, deployment could be implemented as a overlay network which leads again to transactions in overlay networks.

Overlay networks Overlay networks naturally exhibit good properties essentially w.r.t. scalability, availability and autonomicity (mainly self-configuration).

The M2M use case, and more specifically the self-configuration of the log network scenario exhibits a requirement towards self-configuration (the logging overlay nodes should discover each others, determine and maintain routes/connections between each others automatically) and possibly self-dimensioning (the logging overlay could determine itself how many logging nodes with which storage capability it needs according read/write queries it receives). Note that the M2M use case identifies only the implementation of the logging function as self-configured overlay network but other infrastructure services (or 'technical services' or 'non functional services' or 'enablers') could be implemented as self-configured overlay networks as well e.g. notification, deployment, accounting, billing, etc.

A deeper use of overlay networks in the proposed M2M use case would concern a more drastic architectural re-engineering of the current France Telecom M2M platform in which the complete M2M infrastructure (i.e. all types of nodes) would be implemented as an self-configured overlay network which would basically provides for a SON-based trading layer between data producers (e.g. homes in our applicative context) and data consumers (Fire alarming, thermal regulation and other M2M services in our applicative context).

3.1.4 Conclusion

The previous sections described an M2M (Machine-To-Machine) use case that deals with the autonomic management of the thermal environment of buildings. The use case introduced the applicative context, some autonomic scenarios in this context then some high level requirements in terms of desired properties and some associated technical requirements in terms of technical services (or functions) needed to support the identified generic requirements.

Complementary to this synthetic description of the M2M use case, this deliverable contains in the appendices the following material produced by WP5/M2M Use Case in the first year of Selfman:

- The 2 documents: “An Open Component Model and Its Support in Java” and “Fractal Component-Based Software Engineering - Report of Fractal CBSE Workshop at ECOOP’06” provide some background on the Fractal component model which is the preferred architectural framework considered in the M2M autonomic use case.
- The 5 documents: “Towards a Flexible Middleware for Autonomous Integrated Management Applications”, “Composite Probes: a Generic Monitoring Framework for Hierarchical Management of Heterogeneous Data”, “Flexible Reactive Capabilities in Component-Based Autonomic Systems”, “Reliability of Dynamic Reconfigurations in Component-Based Systems” and ‘Performance Characterization of Black Boxes with Self-Controlled load injection for Simulation-based Sizing” describe elements of an generic autonomic framework that is essentially an extension of the Fractal component framework for autonomic computing.

3.2 Distributed Database Use Case

This Section will give a short overview of the wiki(pedia) scenario that illustrates a distributed database system and discusses some of the requirements in the context of the SELFMAN project.

3.2.1 Applicative Context

Wikipedia is a free encyclopedia where any user can edit existing or add new content. The current version is a traditional three-tier application with proxies, application servers and database servers [4].

For SELFMAN, it can be treated as database-like application with versioning, replication and transactions. Trust management and self-optimization can also easily be integrated.

The wiki scenario covers the major topics of SELFMAN and in addition the real data sets are available for testing (http://download.wikipedia.org/backup-index.html) ranging in size from several kilobytes to several gigabytes.

3.2.2 Scenarios

Centralized Environment The architecture as of today follows a traditional three-tier design, where each server serves only a single purpose – database, business logic, or proxy (see Fig 12). The performance of the business logic servers and the proxies can be easily increased by adding more machines as this code is inherently parallel with almost no synchronisation overhead.

The shared state is stored in the database servers. Read as well as write accesses usually require synchronization between the nodes which limits the scalability.

Trusted Environment Instead of using separate hosts as proxies, application and database servers, each wikipedia server could run one peer of a structured overlay with a database like abstraction which provides the same functionality as the existing software.

The self-* components could autonomously balance the load and adapt system parameters to optimize the performance, e.g. the replication factor. Replica placement is also an issue as the performance can be improved by placing German content on servers in the center of Europe.

Untrusted Environment The server farm could be replaced by a structured overlay network where the peers run on the computers of the users of wikipedia. In this case self-* components, replication and trust management become more challenging. ...

Running the system in an untrusted environment means that the system has to handle a higher failure rate among nodes. Replication is needed to guarantee availability of the content in case of node failures. However with replication in SONs, keeping replicas in a consistent state becomes more complicated. In fact it is not possible to optimize for high availability, consistency and network partition

[4]http://meta.wikimedia.org/wiki/Wikimedia_servers

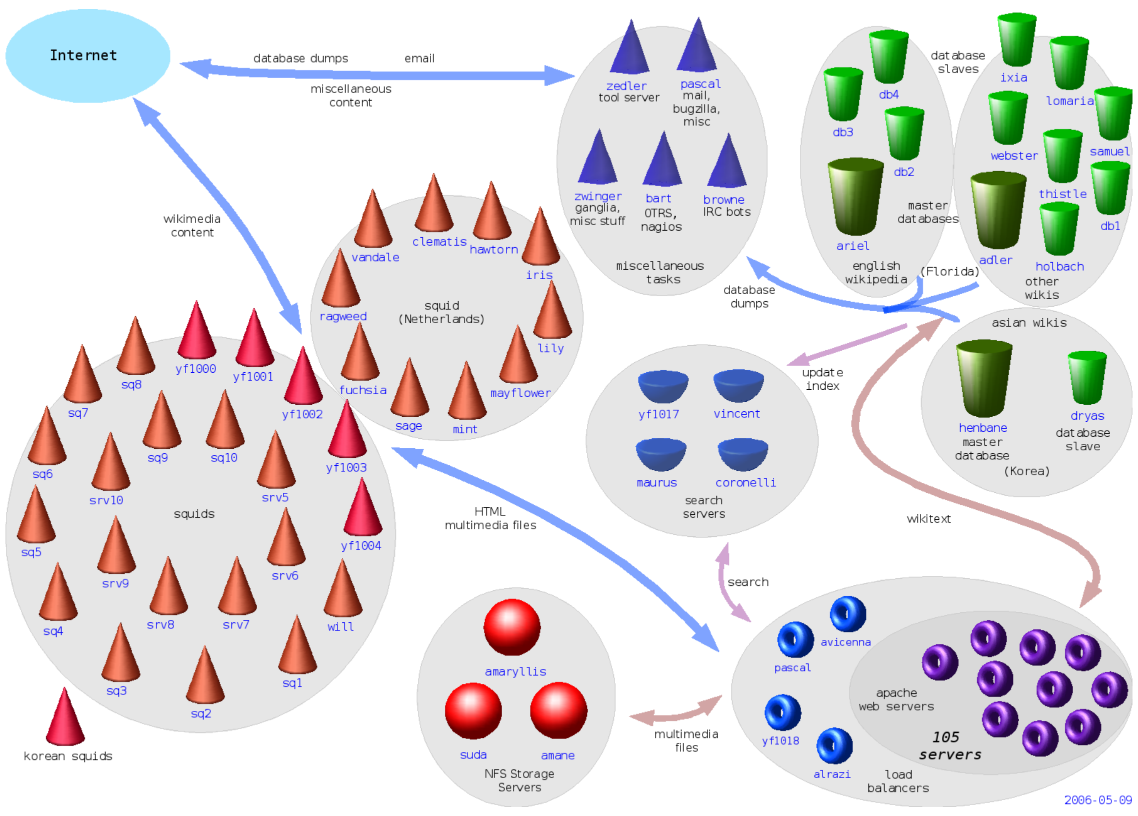

Figure 12: Wikipedia Server Farm as of 2006-05-09.

tolerance at the same time. It has to be investigated to which extent availability should be optimized and which consistency level is sufficient for a wiki scenario.

To provide strong consistency, update operations have to be atomic. We might even do transactions on data. Suppose a user editing some content. The system has to check whether the user had been working on the latest version at the time he wants to make his changes permanent. It should be ensured that changes made in between don't get lost. Data therefore gets some kind of version tags, and transactional mechanisms ensure that changes made by other users are not overwritten.

Besides regular editing and reading of pages, modern Wikis provide a host of additional features: support for attachments, user-based access control, storing per-page metadata (author, geographic coordinates etc.), anti-spam filtering, and several techniques to enhance navigation. These include fulltext searching, backlinks, and page categorization. All navigational features require storing of additional information that must be kept synchronized with page content. For example, to provide backlinks, it is necessary to have an additional reverse index that allows to find all links pointing to a page. Keeping these additional index structures synchronized requires the use of atomic transactions. Without atomicity, it becomes impossible to update page content and index structures consistently.

Update of a Page Editing a single page of wikipedia already requires a simple transaction. When updating a wikipedia page it is first read, before changes are applied on the read content. When the changes should be submitted it has to be

Relation name	Key	Value
Contents	PageName and Version	Contents and Categories
CategoriesIndex	CategoryName	PageName

Table 1: Simple relational model for storing wiki pages and categories

```
updatePage(PageName, Contents, LastVersionSeen)
  BOT //Begin Of Transaction
    if(CurrentVersion(PageName) == LastVersionSeen)
      OldContents = UpdateContent(PageName, Contents)

      OldCategories = ExtractCategories(OldContents)
      NewCategories = ExtractCategories(Contents)

      foreach c in NewCategories - OldCategories
        AddPageToCategory(PageName, c)

      foreach c in OldCategories - NewCategories
        RemovePageFromCategory(PageName, c)
    else
      FailTransaction
    end
  EOT //End Of Transaction
```

Figure 13: Code for updating a page

checked whether the read version is still the current one and whether the changes can be applied. Thereby the version number of the read wikipedia page is compared with the version number of the page stored in the storage system. Additionally an update to a wikipedia page might require updating another page at the same time, e.g. a category page which is related to the particular page. In this case the update on the one page should only take place if it can be done on the other page at the same time. A transaction provides a mechanism to ensure this.

In a very simple model each page has a *name*, a *version number* and belongs to several *categories* (Tab. 3.2.2). To update a page, the current version number of the page in the database has to match the version number of the page the user was editing – there were no updates of the page since the user started to change the page. If check passed the page can be updated in the database and the categories table is updated to reflect the changes.

Searching for category pages In order to retrieve consistent results, searching wikipedia pages requires the use of transaction processing, too. A range query is necessary to query the overlay for all pages that fall in a specific category (Fig. 14).

If it is required that no concurrently added page should be missed by this query, predicate locking is necessary. Besides read-locking the index entries themselves, in this simple case, this requires locking the whole category for insertion and deletion.

```
categoryPages(categoryName)
  BOT //Begin Of Transaction
    pages = GetMatching("CategoriesIndex", "CategoryName=" + categoryName)
    Sort(pages, "Version")
    return pages
  EOT //End Of Transaction
```

Figure 14: Code for searching category pages

3.2.3 Autonomic Scenarios

Self-Optimization: Global Load-Balancing Research on SONs usually assumes that all stored entries are equally popular. Real-world access patterns usually follow a Zipf-distribution, where a few items are very popular and the majority of the items are seldom accessed. In addition the access pattern varies over the day because of the different timezones.

A self-optimizing component is needed which adjusts the mapping of the key space of the SON to the nodes and adjusts the replication factor for individual keys.

Self-Healing: Recovery of Data on Failure This scenario describes the mechanisms needed to handle unexpected node failures. First of all, the integrity of the data stored on the failed node has to be guaranteed. Using redundancy, the database has to implement safety measures to be able to restore the missing data. Using failure detectors neighboring nodes in the overlay will monitor each other and find replacement nodes in cases of failure.

3.2.4 Requirements

This section identifies the requirements and components associated with the scenarios described above. The distributed scenario needs three components:

- Distributed Database
- Distributed Webserver
- Distributed Trust Management

Each of them and the specific requirements are described in the following.

Components

Distributed Database The current backend of wikipedia is a MySQL database distributed over 15 servers where the distribution of data over the server is configured by hand and certain tasks can only be performed by a few and not by all machines (Master-Slave).

A distributed database build on top of a DHT must at least provide the following functionality.

Versioning. Each article is stored with its whole history and there exists a total order over all the versions of an article.

At any time, changes to an article can be reverted and the differences between several versions can be displayed.

When a user updates an article it has to be guaranteed that the changes were made on the latest version, i.e. the user saw the latest version before he started to change the article.

Distributed Webserver A distributed webserver which accesses the database and renders the web pages.

Distributed Trust Management A distributed trust management component to assess the quality of the contributions of users.

Autonomics

Self-Repairing To increase the reliability and availability of the database all information has to be replicated over several nodes to tolerate the failure of small numbers of nodes and repair mechanism are needed to fix failed nodes autonomously.

Self-Optimization The structured overlay has to employ an autonomous load-balancing scheme which takes popularity of items and the influence of time-zones into account.

Transactions A transaction mechanism is needed to ensure atomicity of the required operations for an update. It is also needed to prevent concurrent update from violating the consistency of the content.

Overlay networks There very few requirements to the overlay network which go beyond the features provided by standard SONs. Of importance is only the maintenance overhead in the Trusted Environment scenario. As the failure is very low the ring maintenance should be adapted accordingly.

3.2.5 Conclusion

The main challenge is to build a system based on a SON, which provides the user with an acceptable performance and sufficient data consistency. Therefore a proper trade-off between availability, data consistency and network partition tolerance has to be found.

3.3 P2P Video Streaming Use Case

This section introduces the P2PTV use case proposed par the PeerTV company. The goal of this use case is to be able to distribute live media-streams from any source to large number of consumers (nodes), 100 K to Million nodes, without using any expensive central resources.

3.3.1 Applicative Context

IP multicast is currently mostly disabled by most ISPs due to the extra cost incurred on the routers and incompatibilities between different Autonomous Systems (AS?s). The current approach for live streaming uses large and expensive server equipment allocated nearby the router devices to perform media distribution. An alternative solution is to use overlay networks of consumer nodes to broadcast live-streams, thereby minimizing the cost of deployment and provisioning by individual media providers. On the top of the overlay the nodes will be dynamically organized as multicast trees. Nodes will cooperate in streaming by exploiting their upload bandwidth capacities to multicast streams to other nodes.

3.3.2 Autonomic scenarios

Peer-to-peer live streaming is challenging problem in the context of Selfman as one needs to:

- maximize the total utilization of upload bandwidth,
- minimize latency, and to
- dynamically reconfigure the trees during network dynamism.

The first requirement stresses that the solution needs to ensure that the actual available upload bandwidth at each node should be utilized as much as possible. Any solution must, therefore, adapt to the given upload bandwidth of the individual nodes. This implies that even nodes with petty upload bandwidth should be utilized.

The second requirement puts focus on latencies between the actual nodes. It also implies that the depth of the multicast trees should be shallow to minimize latencies (this is at least true given identical node latencies).

Finally, the solution should continuously reconfigure the system, as there will always be some network dynamism. By network dynamism we refer to i) nodes joining/leaving/failing, or ii) network capacity changing due to network congestion etc.

3.3.3 Requirements

There are a number of requirements on the systems built in Selfman to be able to handle this type of applications:

1. It should be possible to dynamically build multicast trees on the top of the overlays designed in the project.

2. It should be possible to optimize continuously the nodes in the multicast trees so that nodes with higher upload capacities are on the top of the multicast trees and nodes with lower capacities near to the leaves.

3. It should be possible to optimize continuously the nodes in the multicast trees so that the latency between neighboring nodes in the overlays are minimized

4. It should be possible to optimize continuously the nodes in the multicast trees so that the traffic between AS?s are minimized

5. It should be possible to upgrade the software of the overlays remotely and dynsmically. All these requirements pose interesting challenges on the results of Selfman.

3.3.4 Conclusion

Dynamic live-streaming is an interesting application scenario for Selfman as it requires most of the self* properties that the project is set to design on the top of overlay networks.

4 Papers and publications

Appendices of the M2M use case contains the following documents:

- E. Bruneton, T. Coupaye, M. Leclerc, V. Quema, J-B. Stefani. An Open Component Model and Its Support in Java. Published in Software Practice & Experience Journal - Special Issue on Auto-adaptive and Reconfigurable Systems, 36(11-12), 2006.

- T. Coupaye, J-B. Stefani. Fractal Component-Based Software Engineering - Report of Fractal CBSE Workshop at ECOOP'06 . Published in 20th European Conference on Object-Oriented Programming (ECOOP 2006) Workshop Reader, LNCS 4379, 2007.

- M. Kessis, P. Déchamboux, C. Roncancio, T. Coupaye, A. Lefebvre. Towards a Flexible Middleware for Autonomous Integrated Management Applications. Published in 2006 at the International Multi-Conference on Computing in the Global Information Technology (ICCGI'06), August 2006.

- M. Leger, T. Coupaye, T. Ledoux. Reliability of Dynamic Reconfigurations in Component-Based Systems. France Telecom Technical Report, February 2007.

- A. Diaconescu, B. Dillenseger. Composite Probes: a Generic Monitoring Framework for Hierarchical Management of Heterogeneous Data. Submitted for publication by France Telecom in April 2007.

- N. Jayaprakash, T. Coupaye, C. Collet, P.-C. David. Flexible Reactive Capabilities in Component-Based Autonomic Systems. Submitted for publication by France Telecom in May 2007.

- A. Harbaoui, B. Dillenseger, J.-M. Vincent. Performance Characterization of Black Boxes with Self-Controlled load injection for Simulation-based Sizing. Submitted for publication by France Telecom in May 2007.

A The Fractal Component Model and its Support in Java

SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2003; **00**:1–7 *Prepared using* speauth.cls *[Version: 2000/03/16 v2.12]*

The FRACTAL Component Model and Its Support in Java

Eric Bruneton[1], Thierry Coupaye[1], Matthieu Leclercq[2], Vivien Quéma[2], Jean-Bernard Stefani[2]

[1] *France Telecom R&D*
[2] *INRIA Rhône-Alpes*
{Eric.Bruneton,Thierry.Coupaye}@rd.francetelecom.com,
{Matthieu.Leclercq,Vivien.Quema,Jean-Bernard.Stefani}@inrialpes.fr

SUMMARY

This paper presents FRACTAL, a hierarchical and reflective component model with sharing. Components in this model can be endowed with arbitrary reflective capabilities, from plain black-box objects to components that allow a fine-grained manipulation of their internal structure. The paper describes JULIA, a Java implementation of the model, a small but efficient run-time framework, which relies on a combination of interceptors and mixins for the programming of reflective features of components. The paper presents a qualitative and quantitative evaluation of this implementation, showing that component-based programming in FRACTAL can be made very efficient.

KEY WORDS: component-based programming, reflective component model, aspects and components, Java components

1. Introduction

By enforcing a strict separation between interface and implementation and by making software architecture explicit, component-based programming can facilitate the implementation and maintenance of complex software systems [36]. Indeed, these two principles form the basis for two essential properties: adaptability and manageability. Their role as units of software deployment and configuration in particular, are well understood: they allow for pre-run time adaptation in order to suit arbitrary deployment environments (construction of dedicated software infrastructures), evolutions in requirements and technical evolutions (maintenance), and organisational evolutions (integration, interoperation). When seen as run-time structures, components can serve as the basis for software reconfiguration. By fully delineating subsystem boundaries, they provide a natural scope for reconfiguration actions and a natural target for system instrumentation and supervision. Coupled with the use of meta-programming techniques, component-based programming can hide to application programmers some of the complexity inherent in the handling of non-functional aspects in a software system, such as distribution and fault-tolerance, as exemplified by the container concept in Enterprise Java Beans (EJB), CORBA Component Model (CCM), or Microsoft .Net [36].

Existing component-based frameworks and architecture description languages (see e.g. [27] for a recent survey of ADLs), however, provide only limited support for extension and adaptation, as witnessed by recent works on component aspectualization, e.g. [22, 30, 32]. The paper [30] argues at length, for instance, about the lack of tailorability of EJB containers, meaning that there is no mechanism to configure an EJB container or its infrastructural services, nor is it possible to add new services to it.

This limitation implies several important drawbacks: it prevents the easy and possibly dynamic introduction of different control facilities for components such as non-functional aspects; it prevents application designers and programmers from making important trade-offs such as degree of configurability vs performance and space consumption; and it can make difficult the use of these frameworks and languages in different environments, including embedded systems. Even with reflective component models such as OpenCOM [20], i.e. models of components endowed with an explicit meta-object protocol to control the execution of components and introduce support for different non-functional aspects, we find it necessary to be able to finely tailor the reflective capabilities endowed in components in order to support the different trade-offs which are critical in low-level software infrastructure design, e.g. in operating system or middleware construction.

We present in this paper a component model, called FRACTAL, that alleviates the above problems by introducing a notion of component endowed with an open set of control capabilities. In other terms, components in FRACTAL are reflective, in the sense that their execution and their internal structure can be made explicit and controlled through well-defined interfaces. These reflective capabilities, however, are not fixed in the model but can be extended and adapted to fit the programmer's constraints and objectives.

Importantly, we also present in this paper how such an *open* component model can be efficiently supported in Java by an extensible run-time framework, called JULIA. JULIA incorporates an innovative use of mixin classes to allow the definition and combination of arbitrary component controllers. This provides a JULIA programmer with effective and extensible means to deal with different cross-cutting aspects in controlling components.

The main contributions of the paper are as follows:

- We define a hierarchical component model with sharing, that supports an extensible set of component control capabilities.
- We show how this model can be effectively supported in Java by means of an extensible software framework, that provides for both static and dynamic configurability.
- We show that our component model and run-time framework can be used effectively to build highly configurable, yet efficient, distributed systems.

The paper is organized as follows. Section 2 presents the main features of the FRACTAL model. Section 3 describes JULIA, a Java framework that supports the FRACTAL model. Section 4 evaluates the model and its supporting framework. Section 5 discusses related work. Section 6 concludes the paper with some indications for future work.

2. The Fractal component model

The Fractal component model (see [18] for a detailed specification), is a general component model which is intended to implement, deploy and manage (i.e. monitor, control and dynamically configure) complex software systems, including in particular operating systems and middleware. This motivates the main features of the model:

- Composite components (components that contain sub-components), in order to have a uniform view of applications at various levels of abstraction.
- Shared components (sub-components of multiple enclosing composite components), in order to model resources and resource sharing while maintaining component encapsulation.
- Introspection capabilities, in order to monitor and control the execution of a running system.
- Re-configuration capabilities, in order to deploy and dynamically configure a system.

To allow programmers to tune the control of reflective features of components to the requirements of their applications, Fractal is defined as an extensible system. Control features of components are not predetermined in the model, rather the model allows for a continuum of reflective features or *levels of control*, ranging from no control (black-boxes, standard objects) to full-fledged introspection and intercession capabilities (including e.g. access and manipulation of component contents, control over components life-cycle and behavior, etc).

2.1. Components and bindings

A Fractal component is a run-time entity that is encapsulated, has a distinct identity, and that supports one or more interfaces. An *interface* is an access point to a component (similar to a "port" in other component models), that implements an *interface type* (i.e. a type specifying the operations supported by the access point). Interfaces can be of two kinds: server interfaces, which correspond to access points accepting incoming operation invocations, and client interfaces, which correspond to access points supporting outgoing operation invocations.

A Fractal component (see Figure 1) can be understood generally as being composed of a *membrane*, which supports interfaces to introspect and reconfigure its internal features, and a *content*, which consists in a finite set of other components (called *sub-components*). The membrane of a component can have external and internal interfaces. External interfaces are accessible from outside the component, while internal interfaces are only accessible from the component's sub-components. The membrane of a component is typically composed of several controllers. Typically, a membrane can provide an explicit and causally connected representation of the component's sub-components and superpose a control behavior to the behavior of the component's sub-components, including suspending, checkpointing and resuming activities of these sub-components. Controllers can also play the role of interceptors. Interceptors are used to export the external interface of a subcomponent as an external interface of the parent component. They can intercept the oncoming and outgoing operation invocations of an exported interface and they can add additional behavior to the handling of

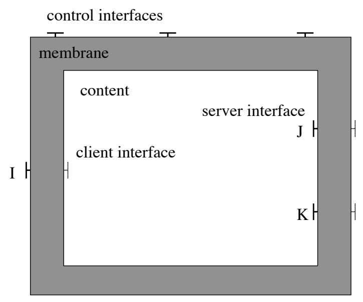

Figure 1. A Fractal component

such invocations (e.g. pre and post-handlers). Each component membrane can thus be seen as implementing a particular semantics of composition for the component's sub-components. Controller can be understood as meta-objects or meta-groups as they appear in reflective languages and systems.

The FRACTAL model provides two mechanisms to define the architecture of an application: component nesting we have just described and bindings. Communication between FRACTAL components is only possible if their interfaces are bound. FRACTAL supports both primitive bindings and composite bindings. A *primitive binding* is a binding between one client interface and one server interface in the same address space (which can be modeled as a component), which means that operation invocations emitted by the client interface should be accepted by the specified server interface. A primitive binding is called that way for it can be readily implemented by pointers or direct language references (e.g. Java references). A *composite binding* is a communication path between an arbitrary number of component interfaces. These bindings are built out of a set of primitive bindings and binding components (stubs, skeletons, adapters, etc). A binding is a normal FRACTAL component whose role is to mediate communication between other components. The binding concept corresponds to the connector concept that is defined in other component models. Note that, except for primitive bindings, there is no predefined set of bindings in FRACTAL. In fact bindings can be built explicitly by composition, just as other components. Importantly, bindings can embody remote communication paths between interfaces, and span different address spaces and different machines in a network. This allows the construction of distributed configuration of FRACTAL components.

An original feature of the FRACTAL component model is that a given component can be included in several other components. Such a component is said to be *shared* between these components. Consider, for example, a menu and a toolbar components (see Figure 2), with an "undo" toolbar button corresponding to an "undo" menu item. It is natural to represent the menu items and toolbar buttons as sub components, encapsulated in the menu and toolbar components, respectively. But, without sharing, this solution does not work for

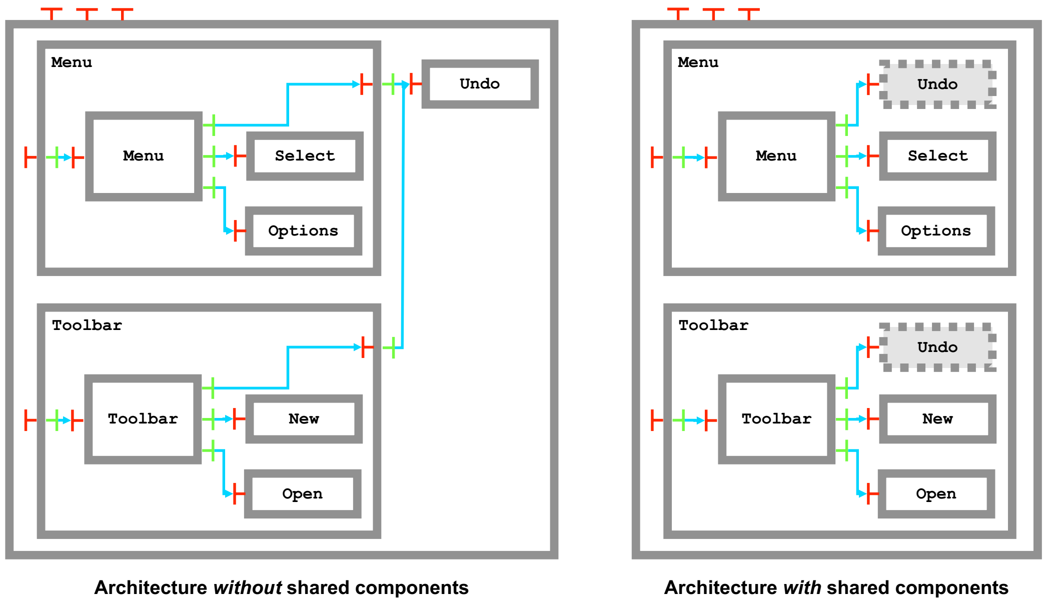

Figure 2. Component sharing in Fractal

the "undo" button and menu item, which must have the same state (enabled or disabled): these components, or an associated state component, must be put outside the menu and toolbar components. With component sharing, the state component can be shared between the menu and toolbar components, in order to preserve component encapsulation. Shared components are also useful to faithfully model access to low-level system resources (which are typically shared between applications), and to help separate "aspects" in component based applications (for instance it is possible to have components representing address spaces, i.e. a physical architecture, with sub components shared with other components representing a logical architecture)

2.2. Levels of control

The Fractal model does not enforce a fixed and pre-determined set of controllers in component membranes (hence the phrase "*open* component model"). It allows instead arbitrary forms of membranes, with different control and interception semantics. The Fractal specifiation, however, identifies specific forms of membranes and controllers, corresponding to different levels of control (or reflection capabilities) on components.

At the lowest level of control, a Fractal component is a black box, that does not provide any introspection or intercession capability. Such components, called *base components*, are comparable to plain objects in an object-oriented programming language such as Java (although, even at the lowest level of control, the model allows components to have a varying number of interfaces during their lifetime). Their explicit inclusion in the model facilitates the integration of legacy software.

At the next level of control, a FRACTAL component provides a Component interface, similar to the IUnknown in the COM model, that allows one to discover all its external (client and server) interfaces. Each interface has a name that distinguishes it from other interfaces of the component. At this level of control, components still do not provide any control function, but the Component interface provides elementary means for introspecting the external structure of a component. Also at this level of control, component interfaces may additionally support, via multiple interface inheritance, operations that allow to retrieve the Component interface of the supporting component. Such operations are gathered in the Interface interface type.

At upper levels of control, a FRACTAL component can expose elements of its internal structure, and provide increased introspection and intercession capabilities. The FRACTAL specification provides several examples of useful forms of controllers, which can be combined and extended to yield components with different reflective features:

- **Attribute controller:** An attribute is a configurable property of a component. A component can provide an AttributeController interface to expose getter and setter operations for its attributes.
- **Binding controller:** A component can provide the BindingController interface to allow binding and unbinding its client interfaces to server interfaces by means of primitive bindings.
- **Content controller:** A component can provide the ContentController interface to list, add and remove subcomponents in its contents.
- **Life-cycle controller:** A component can provide the LifeCycleController interface to allow explicit control over its main behavioral phases, in support for dynamic reconfiguration. Basic lifecycle methods supported by a LifeCycleController interface include methods to start and stop the execution of the component.

2.3. Type system

The FRACTAL model is endowed with an optional type system (some components such as base components need not adhere to the type system). Interface types describe the operations supported by an interface, the role of the interface (client or server), as well as its *contingency* and its *cardinality*. The contingency of an interface indicates if the functionality corresponding to this interface is guaranteed to be available or not, while the component is running:

- The operations of a *mandatory* interface are guaranteed to be available when the component is running. For a client interface, this means that the interface must be bound. As a consequence, a component with mandatory client interfaces cannot be started until all these interfaces are bound.
- The operations of an *optional* interface are not guaranteed to be available. For a server interface, this can happen e.g. when the complementary internal interface of the

supporting component is not bound to a sub-component interface. For a client interface, this means that the component can execute without this interface being bound.

The cardinality of an interface type T specifies how many interfaces of type T a given commponent may have. A *singleton* cardinality means that a given component must have exactly one interface of type T. A *collection* cardinality means that a given component may have an arbitrary number of interfaces of type T. Such interfaces are typically created lazily, e.g. upon request of a bind operation through a BindingController interface.

Component types are just sets of component interface types. The type system is equipped with a subtyping relation which embodies constraints to ensure substitutability of components.

2.4. Instantiation

The FRACTAL model also defines *factory* components, i.e. components that can create new components. Again, the FRACTAL model does not constrain the form and nature of factory components, but the FRACTAL specification provides useful forms of such factories. In particular, it distinguishes between *generic component factories*, which can create several kinds of components, and *standard factories*, which can create only one kind of components, all with the same component type. A generic factory provides the GenericFactory interface, which allows a new component to be created, given its type, and an appropriate description of its membrane (controllers) and content. A *template* is a special standard factory that creates components that have the same internal structure as the template. Thus, a template component can have several templates as sub-components (sub-templates). A component created by such a template will have as many sub-components as sub-templates in the template, which will be bound together in the same way as the sub-templates are. Templates are useful to manifest at run-time a particular configuration.

3. The JULIA framework

The JULIA framework supports the construction of software systems with FRACTAL components written in Java. The main design goal for JULIA was to implement a framework to program FRACTAL component membranes. In particular, we wanted to provide an extensible set of control objects, from which the user can freely choose and assemble the controller and interceptor objects he or she wants, in order to build the membrane of a FRACTAL component. The second design goal was to provide a continuum from static configuration to dynamic reconfiguration, so that the user can make the speed/memory tradeoffs he or she wants. The last design goal was to implement a framework that can be used on any JVM and/or JDK, including very constrained ones such as the KVM, and the J2ME profile (where there is no ClassLoader class, no reflection API, no collection API, etc). In addition to the previous design goals, we also made two hypotheses in order to simplify the implementation: we suppose there is only one (re)configuration thread at a given time, and we also suppose that the component data structures do not need to be protected against malicious components.

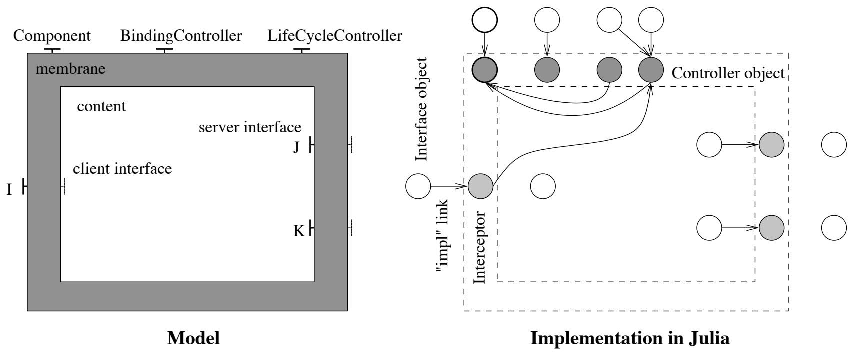

Figure 3. An abstract component and a possible implementation in JULIA

3.1. Main Data Structures

A FRACTAL component is generally represented by many Java objects, which can be separated into three groups (see Fig. 3):

- the objects that implement the component interfaces, in white in Fig. 3 (one object per component interface; each object has an impl reference to an object that really implements the Java interface, and to which all method calls are delegated; this reference is null for client interfaces; for server interfaces it can reference an interceptor or an object of the content part),
- the objects that implement the membrane of the component, in gray and light gray in the figure (a controller object can implement zero of more control interfaces),
- and the objects that implement the content part of the component (not shown in the figure).

The fact that each component interface is represented by its own Java object comes from the fact that component interfaces are typed (i.e., a component interface object implements both Interface and the Java interface corresponding to this interface). It is not possible to do better, unless perhaps by using very complex bytecode manipulations that modify the signature of all the methods of all classes.

The objects that represent the membrane of a component can be separated into two groups: the objects that implement the control interfaces (in gray in Fig. 3), and (optional) *interceptor* objects (in light gray) that intercept incoming and/or outgoing method calls on non-control interfaces. These objects implement respectively the Controller and the Interceptor interfaces. Each controller and interceptor object can contain references to other controller / interceptor objects (since the control aspects are generally not independent - or "orthogonal" - they must generally communicate between each other).

Instantiation

JULIA components can be created manually or automatically. The manual method can be used to create any kind of components, while the automatic one is restricted to components whose type follows the basic type system defined in the FRACTAL specification, which provide a Component interface, and which provide interface introspection functions. In both methods, a component must be created as follows:

- creation of the component interface objects (if the component must provide interface introspection), of the controller objects, of the interceptor objects, and of the component's content (for container components).
- initialization of the impl references between the component interfaces objects and the content, controller and interceptor objects.
- creation of an InitializationContext, and set up of this context, with references to the previous objects.
- initialization of the controller and interceptor objects by calling their initFcController method, with the previous InitializationContext object as parameter (this step allows the controller and interceptor objects to initialize themselves, i.e. to set up the references between all these objects).

In the automatic method, i.e. when components are created through the GenericFactory interface, the operations that must be done at the previous steps are deduced from the component's type, and from its membrane and content descriptor. Once these descriptors have been analyzed and checked, and once the previous operations have been determined, a sub class of the InitilizationContext class that implements these operations is generated, directly in bytecode form, with the InitializationContextClassGenerator. Finally the component is created by using this generated class (in other words, the controller and content descriptors are compiled on the fly, once and for all, instead of being interpreted and checked each time a component must be created).

3.2. Mixin classes

3.2.1. Motivations

The main design goal of JULIA is to implement a reusable and extensible *framework* to program component membranes. In particular, since everything in the Fractal specification is optional, JULIA must provide implementations of the Fractal API interfaces for any conformance level. For example, JULIA must provide a basic Component implementation, as well as an implementation for components whose type follows the basic type system (in the first case Component behaves like a read only hash map; in the second case, because of collection interface types, the getFcInterface method can lazily create new component interfaces - see section 2.3). Likewise, JULIA must provide a basic BindingController implementation, as well as an implementation for cases where the basic type system is used, where a life cycle controller is present, or where composite components are used (these implementations are needed to check type, life cycle or content related constraints on bindings). There must also be an implementation for cases where both the basic type system and a life cycle controller are used, or where the basic type system, life cycle controllers, and composite components are used. And these implementations must be extensible, in order to take into account user defined controllers when needed.

In order to provide all these implementations, a first solution would be to use class inheritance. But this solution is not feasible, because it leads to a combinatorial explosion, and to a lot of code duplication. Consider for example the BindingController interface, and the "type system", "life cycle" and "composite" concerns. These three concerns give $2^3 = 8$ possible combinations. Therefore, in order to implement these three concerns, *eight* classes (and not just three) must be provided. Moreover these eight classes can not be provided without duplicating code, if multiple inheritance is not available.

Another solution to this problem would be to use an Aspect Oriented Programming (AOP) tool or language, such as Aspect/J [24], since the goal of these tools and languages is to solve the "crosscutting" problems. Aspect/J, for example, could effectively be used to solve the above problem: aspect classes could indeed be used instead of sub classes, which would solve the combinatory problems (an aspect can be applied to multiple classes and aspects). But using Aspect/J would introduce a new problem, due to the fact that, in Aspect/J, 1) aspects must be applied at compile time, and 2) this process requires the source code of the base classes[*]. It would then be impossible to distribute JULIA in compiled form, because then users would not be able to apply new aspects to the existing JULIA classes (in order to add new control aspects that crosscut existing ones).

What is needed to really solve our modularity and extensibility problem is therefore a kind of AOP tool or language that can be used at load time or at runtime, without needing the source code of the base classes, such as JAC [29]. The current JULIA version does not use JAC or other similar systems: it uses instead some kind of *mixin* classes. A mixin class is a class whose super class is specified in an abstract way, by specifying the minimum set of fields and methods it should have. A mixin class can therefore be *applied* (i.e. override and add methods) to any super class that defines at least these fields and methods. This property solves the above combinatory problem. Moreover, mixin classes used in JULIA can be mixed at load time, thanks to our bytecode generator ASM [1] (unlike in AspectJ and in most mixin based inheritance languages, where mixed classes are declared at compile time).

3.2.2. Implementation

Instead of using a Java extension to program the mixin classes, which would require an extended Java compiler or a pre processor, mixin classes in JULIA are programmed by using patterns. For example the JAM [15] mixin class shown below (on the left) is written in pure Java as follows (on the right):

[*]This is no longer true with version 1.1 of AspectJ, but this was the case in 2002 when JULIA was developed.

```
mixin A {
  inherited public void m ();
  public int count;
  public void m () {
    ++count;
    super.m();
  }
}
```

```
abstract class A {
  abstract void _super_m ();
  public int count;
  public void m () {
    ++count;
    _super_m();
  }
}
```

In other words, the _super_ prefix is used to denote the inherited members in JAM, i.e. the members that are required in a base class, for the mixin class to be applicable to it. More precisely, the _super_ prefix is used to denote methods that are overridden by the mixin class. Members that are required but not overridden are denoted with _this_:

```
abstract class M implements I {
  abstract void _super_m ();
  abstract void _this_n ();

  public int count;
  public void m () {
    ++count;
    _this_n();
    _super_m();
  }
}
```

Mixin classes can be mixed, resulting in normal classes. More precisely, the result of mixing several mixin classes M_1, ... M_n, *in this order*, is a normal class that is equivalent to a class M_n extending the M_{n-1} class, itself extending the M_{n-2} class, ... itself extending the M_1 class (constructors are ignored; an empty public constructor is generated for the mixed classes). Several mixin classes can be mixed only if each method and field required by a mixin class M_i is provided by a mixin class M_j, with $j < i$ (each required method and field may be provided by a different mixin). For example, if N and O designate the following mixins:

```
abstract class N implements I {
  abstract void _super_m ();

  public void m () {
    System.out.println("m called");
    _super_m();
  }
}
```

```
abstract class O implements I {
  public void m () {
    System.out.println("m");
  }
  public void n () {
    System.out.println("n");
  }
}
```

then the mixed class O N M is equivalent to the following class (note that this class implements all the interfaces implemented by the mixin classes):

```
public class C55d992cb_0 implements I, Generated {
  // from O
  private void m$1 () {
    System.out.println("m");
  }
  public void n () {
    System.out.println("n");
  }
  // from N
  private void m$0 () {
    System.out.println("m called");
    m$1();
  }
  // from M
  public int count;
  public void m () {
    ++count;
    n();
    m$0();
  }
}
```

The mixed classes are generated dynamically, directly in bytecode form with ASM [1], by the MixinClassGenerator class. In order to ease debugging, the class generator keeps the line numbers of the mixin classes in the mixed class. More precisely, a line number l of the mixin class at index i (in the list of mixin classes, and starting from 1) is transformed into $1000*i+l$. For example, if a new Exception().printStackTrace() were added in the N.m method, the stack trace would contain a line at C55d992cb_0.m$0(ONM:2005), meaning that the exception was created in method m$0 of the C55d992cb_0 class, whose source is the ONM mixed class, at line 5 of mixin 2, i.e. at line 5 of the N mixin class.

3.3. Interceptors

Some control aspects, such as the control of bindings, can be completely implemented in a generic way, in a single controller object. But most control aspects must be implemented in two parts: a generic part, and a non generic "hook" part that must be weaved into the user code. In JULIA, this non generic "hook" part is made of the interceptor objects (see Section 3.1), and the weaving is done by inserting these interceptor objects between user objects.

The interceptor classes, since they are not generic (they must implement one or more user interfaces), cannot be written by hand, unlike controller classes, and must therefore be generated automatically. As in some Meta Object Protocol (MOP) implementations, JULIA generates these classes directly in compiled form, by using the ASM library. This is much faster than generating these classes in source code form, which allows JULIA to generate them dynamically, during the application's execution (but JULIA also offers the possibility to generate them statically, before launching an application).

However, unlike in most MOP implementations, the generator that generates the interceptor classes is open and extensible. This flexibility was introduced for efficiency reasons. Indeed this open generator can be used not only to generate interception code that reifies all method calls, as in MOPs, but also to generate much more efficient code, specialized for a given set of aspects. This open generator is described in the rest of this section.

The interceptor class generator takes as parameters the name of a super class, the name(s) of one or more application specific interface(s), and one or more aspect code weaver(s). It generates a sub class of the given super class that implements all the given application specific interfaces and that, for each application specific method, implements all the aspects corresponding to the given aspect code weavers.

Each aspect code weaver is an object that can manipulate the *bytecode* of each application specific method *arbitrarily*. For example, an aspect code weaver A can modify an empty interception method `void m () { return delegate.m() }` into:

```
void m () {
  // pre code A
  try {
    delegate.m();
  } finally {
    // post code A
  }
}
```

where the pre and post code blocks can be adapted to the precise arguments and return types of `m`, while another aspect code generator B will modify this method into:

```
void m () {
  // pre code B
  delegate.m();
}
```

When an interceptor class is generated by using several aspect code weavers, the transformations performed by these weavers are automatically composed together. For example, if A and B are used to generate an interceptor class, the result for the previous `m` method is the following (depending on the order in which A and B are composed):

```
void m () {               void m () {
  // pre code A             // pre code B
  try {                     // pre code A
    // pre code B           try {
    delegate.m();             delegate.m();
  } finally {               } finally {
    // post code A            // post code A
  }                         }
}                         }
```

Note that, thanks to this elementary automatic weaving, which is very similar to what can be found in Aspect/J, several aspects can be managed by a single interceptor object: there is no need to have chains of interceptor objects, each object corresponding to an aspect.

Like the controller objects, the aspects managed by the interceptor objects of a given component can all be specified by the user when the component is created. The user can therefore not only choose the control interfaces he or she wants, but also the interceptor objects he or she wants.

JULIA provides two specific aspect code weavers (to manage the life cycle and trace aspects), and two generic code weavers that reify method calls (one that just reifies method names, and one that also reifies the arguments). Users can of course provide their own code weavers, but writing such a weaver requires a good knowledge of the Java bytecode instructions.

3.4. Optimizations

3.4.1. Intra component optimizations

In order to save memory, JULIA provides some optimization options to merge some or all the objects that make up a component into a single Java object. These optimizations are based on a tool provided by JULIA, which uses the ASM library to merge several controller classes into a single class. This tool is based on the following assumptions:

- each controller object can provide and require zero or more Java interfaces. The provided interfaces must be implemented by the object, and there must be one field per required interface, whose name must begin with weaveable for a mandatory interface, or weaveableOpt for an optional interface (see below). Each controller class that requires at least one interface must also implement the Controller interface (see below).
- in a given configuration, a given interface cannot be provided by more than one object (except for the Controller interface). Otherwise it would be impossible to merge these objects (an object cannot implement a given interface in several ways).
- the bindings between objects in a given configuration are established automatically in a two steps process: a) each controller object of the configuration is registered into a "naming service" (in practice, this naming service is the InitializationContext interface), and b) each controller object initializes itself by using the previous "naming service" to retrieve the interface it requires.

To be more precise, lets suppose we have four control interfaces I, J, K and L, and three controller classes IImpl, JImpl and KImpl. These classes should look like this:

```
public class IImpl implements Controller, I {
  public J weaveableJ;    // = required interface of type J
  public L weaveableOptL; // = optional required interface of type L
  public int foo;         // normal field
```

```
    // implementation of the Controller interface
    public void initFcController (InitializationContext ic) {
      weaveableJ = (J)ic.getInterface("j");
      weaveableOptL = (L)ic.getOptionalInterface("l");
    }
    // other methods
    public void foo (String name) {
      weaveableJ.bar(weaveableOptL, foo, weaveableC.getFcInterface(name));
    }
  }

  public class JImpl implements Controller, J {
    public K weaveableK;
    public void initFcController (InitializationContext ic) {
      weaveableK = (K)ic.getInterface("k");
    }
    // other methods ...
  }

  public class KImpl implements Controller, K {
    // other methods ...
  }
```

In the non optimized case, a component with these three controller objects is instantiated in the following steps. First an instance of IImpl, JImpl and KImpl is created, then the resulting objects are put in an InitializationContext object, and finally the initFcController method is called on each controller object with this context as argument. In the optimized case, the class obtained by "merging" (see below) the IImpl, JImpl and KImpl is dynamically generated (or loaded from the classpath if it has been statically generated before launching the application, or just returned if it has already been generated or loaded) and then an instance of this class is created.

The "merging" process is the following. Basically, all the methods and fields of each class are copied into a new class (the resulting class does not depend on the order into which the classes are copied). However the fields whose name begins with weaveable are replaced with this, and those whose name begins with weaveableOpt are replaced either with this, if a class that implements the corresponding type is present in the list of the classes to be merged, or null otherwise. Finally, the initFcController methods from the Controller interface are merged into a single initFcController method. The result is the following class:

```
  public class Cb234f2 implements Controller, I, J, K, ..., Generated {
    // fields and methods copied from IImpl:
    public int foo;
    public void foo (String name) {
      bar(null, foo, this.getFcInterface(name));
    }
  private void initFcController$0 (InitializationContext ic) {
    (J)ic.getInterface("j");
    (L)ic.getOptionalInterface("l");
  }
  // fields and methods copied from JImpl (not shown) ...
  // fields and methods copied from KImpl (not shown) ...
  // merged initFcController method:
  public void initFcController (InitializationContext ic) {
    initFcController$0(ic);
    initFcController$1(ic);
    initFcController$2(ic);
  }
}
```

As explained in section 3.1, the membrane of a component is made of controller objects and of interceptor objects. The above optimization only applies to controller objects. Therefore, even with this optimization, the membrane of a component is still made, in general, of several Java objects. However, if the interceptor objects all delegate to the same object, and if they do not have conflicting interfaces, it is possible to really have only one Java object for the whole membrane of the component. In this case, which happens for most primitive components, the instantiation process is the following:

- a class that merges the controller classes is generated or loaded as before,
- a sub class of this class that implements the interception code for each method of each functional interface is generated or loaded,
- this sub class is instantiated.

Even if the controllers and interceptors are merged into a single object, the content of the component is still made of a separate object. It is however possible to instantiate a whole component (i.e. the controllers, the interceptors and the content part) as a single Java object. In order to do this, the user component class is used as a super class to generate the merged controller class, which is itself used a super class to generate the interceptor class (as described above).

3.4.2. Inter component optimisations

In addition to the previous intra component optimizations, which are mainly used to save memory, JULIA also provides an inter component optimization, namely an algorithm to create and update *shortcut* bindings between components, and whose role is to improve time performances. As explained in section 3.1, each interface of a component contains an impl reference to an object that really implements the component interface. In the case of a server interface *s*, this field generally references an interceptor object, which itself references another server interface.

More precisely, this is the case with the CompositeBindingMixin. With the OptimizedCompositeBindingMixin, the impl references are optimized when possible. For example, in Fig.

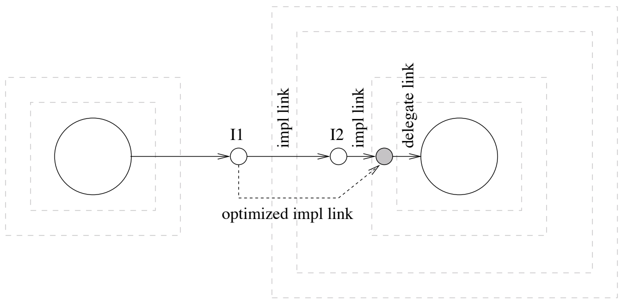

Figure 4. Shortcut bindings

4, since I1 does not have an associated interceptor object, and since component interface objects such as I2 just forward any incoming method calls to the object referenced by their impl field, I1 can, and effectively references directly the interceptor associated to I2. The OptimizedCompositeBindingMixin automatically manages these shortcuts. In particular, this mixin invalidates and recomputes the necessary shortcuts when a binding is modified (indeed, modifying a binding somewhere may invalidate existing shortcuts, and/or create new shortcuts).

3.5. Support for Constrained Environments

One of the goals of JULIA is to be usable even with very constrained JVMs and JDKs, such as the KVM and the J2ME libraries (CLDC profile). This goal is achieved thanks to the following properties.

- The size of the JULIA runtime (35kB, plus 10kB for the FRACTAL API), which is the only part of JULIA (175 kB as a whole) that is needed at runtime, is compatible with the capabilities of most constrained environments.
- JULIA can be used in environments that do not provide the Java Reflection API or the ClassLoader class, which are needed to dynamically generate the JULIA application specific classes, since these classes can also be generated statically, in a less constrained environment.
- The JULIA classes that are needed at runtime, or whose code can be copied into application specific runtime classes, use only the J2ME, CLDC profile APIs, with only two exceptions for collections and serialization. For collections a subset of the JDK 1.2 collection API is used. This API is not available in the CLDC profile, but a bytecode modification tool is provided with JULIA to convert classes that use this subset into classes that use the CLDC APIs instead. This tool also removes all serialization related code in JULIA. In other words the JULIA jars cannot be used directly with CLDC, but can be transformed automatically in new jars that are compatible with this API.

4. Evaluation

We provide in this section an evaluation of our model and its implementation. We first provide a qualitative assessment of our component framework. We then provide a more quantitative evaluation with micro-benchmarks and with an application benchmark based on a reengineered message-oriented middleware.

4.1. Qualitative assessment

Modularity JULIA provides several mixins for the binding controller interface, two implementations of the life cycle controller interface, and one implementation of the content controller interface. It also provides support to control component attributes, and to associate names to components. All these aspect implementations, which make different flexibility/performance tradeoffs, are well separated from each other thanks to mixins, and can therefore be combined freely. Together with the optimization mechanisms used in JULIA, this flexibility provides what we call a *continuum* from static to dynamic configurations, i.e., from unreconfigurable but very efficient configurations, to fully dynamically reconfigurable but less efficient configurations (it is even possible to use different flexibility/performance tradeoffs for different parts of a single application).

Extensibility Several users of JULIA have extended it to implement new control aspects, such as transactions [33], auto-adaptability [21], or checking of the component's behavior, compared to a formal behavior, expressed for example with assertions (pre/post conditions and invariants), or with more elaborate formalisms, such as temporal logic [34]. As discussed below, we have also built with JULIA a component library, called DREAM, for building message-oriented middleware (MOM) and reengineered an existing MOM using this library. DREAM components exhibit specific control aspects, dealing with on-line deployment and re-configuration. In all these experiences, the different mechanisms in JULIA have proved sufficient to build the required control aspects.

Accessibility Besides JULIA, several tools are available to easily implement, assemble, deploy and manage Fractal components in Java: Fractlet provides annotations to generate several artifacts from a single source file (like XDoclet), Fractal ADL can be used to describe and deploy Fractal architectures, Fractal GUI can be used to graphically edit Fractal ADL XML files, and Fractal Explorer and Fractal JMX can be used to introspect and manage running Fractal applications.

Limitations There are however some limitations to JULIA's modularity and extensibility. For example, when we implemented JULIA, it was sometimes necessary to refactor an existing method into two or more methods, so that one of this new methods could be overridden by a new mixin, without overriding the others. In other words, the mixin mechanism is not sufficient by itself: the classes must also provide the appropriate "hooks" to apply the mixins. And it is not easy, if not impossible, to guess the hooks that will be necessary for future aspects (but this problem is not specific to mixins, it also occurs in AspectJ, for example).

options	memory overhead (bytes)	time overhead (μs)
lifecycle, no optimization	592	0.110
lifecycle, merge controllers	528	0.110
lifecyle, merge all	504	0.092
no lifecycle, no optimization	496	0.011
no lifecycle, merge controllers	440	0.011
no lifecycle, merge all	432	0.011

Table I. JULIA performances

4.2. Quantitative evaluation I: Micro-benchmarks

In order to measure the memory and time overhead of components in JULIA, compared to objects, we measured the memory size of an object, and the duration of an empty method call on this object, and we compared these results to the memory size of a component [†] (with a binding controller and a life cycle controller) encapsulating this object, and to the duration of an empty method call on this component. The results are given in Table I, for different optimization options. The measurements were made on a Pentium III 1GHz, with the JDK1.3, HotSpotVM, on top of Linux. In these conditions the size of an empty object is 8 bytes, and an empty method call on an interface lasts 0.014 μs.

As can be seen the class merging options can reduce the memory overhead of components (merging several objects into a single one saves many object headers, as well as fields that were used for references between these objects). The time overhead without interceptor is of the order of one empty method call: it corresponds to the indirection through a component interface object. With a life cycle interceptor, this overhead is much greater: it is mainly due to the execution time of two `synchronized` blocks, which are used to increment and decrement a counter before and after the method's execution. This overhead is reduced in the "merge all" case, because an indirection is saved in this case. In any cases, this overhead is much smaller than the overhead that is measured when using a generic interceptor that completely reifies all method calls (4.6 μs for an empty method, and 9 μs for an `int inc (int i)` method), which shows the advantages of using an open and extensible interceptor code generator.

The time needed to instantiate a component encapsulating an empty object is of the order of 0.3 ms, without counting the dynamic class generation time, while the time to needed instantiate an empty object is of the order of 0.3 μs (instantiating a component requires to instantiate several objects, and many checks are performed before instantiating a component).

[†]the size of the objects that represent the component's type, which is shared between all components of the same type, is not taken into account here. This size is of the order of 1500 bytes for a component with 6 interfaces.

4.3. Quantitative evaluation II: the DREAM communication framework

In this section we present DREAM, a framework for the construction of asynchronous middleware, which relies on JULIA. We first briefly describe the framework. Then we show how it has been used to re-engineer JORAM [3], an open-source JMS-compliant middleware (Java Messaging Service [2]).

4.3.1. A component-based framework for asynchronous middleware

Motivations The use of asynchronous middleware (MOM for *Message-Oriented Middleware*) is recognized as a means of achieving scalability in applications made of loosely coupled autonomous components that communicate on large-scale networks [16]. Several MOMs have been developed in the past ten years [3, 19, 35, 37]. The research work has primarily focused on the support of various non functional properties like message ordering, reliability, security, etc. Less emphasis has been placed on the MOM configurability. Indeed, existing middleware are not very configurable, both at the functional and non-functional level. From the functional point of view, they implement a fixed programming interface (API), thus providing a fixed subset of asynchronous communication models (publish/subscribe, event/reaction, message queues, etc.). From the non-functional point of view, existing middleware often provide the same non-functional properties for all event disseminations. This reduces their performance and makes them difficult or impossible to use with devices having limited computational resources.

To overcome these limitations, we have developed DREAM (*Dynamic REflective Asynchronous Middleware*), a software framework dedicated to the construction of asynchronous middleware. DREAM provides a component library and a set of tools to build, configure and deploy middleware implementing various asynchronous communication paradigms: message passing, event-reaction, publish-subscribe, etc.

Architecture of a DREAM *component* DREAM components are standard Fractal components with two characteristic features: the presence of input/output interfaces and the ability to manipulate DREAM resources (messages and activities).

Input/Output interfaces allow DREAM components to exchange messages. Messages are always sent from outputs to inputs (Figure 5 (a)). Output and input interfaces come in pairs corresponding to two kinds of connections, *push* and *pull*. As shown in Figure 5 (b) and (c), "input" and "output" are roles played by normal client and server interfaces (the input and output roles are played by server and client interfaces, respectively, for a push connection; and vice versa for a pull connection).

Message managers Messages are managed by dedicated shared components, called *message manager*. They allow DREAM components to create, duplicate or delete messages. Messages are particular Fractal composites that encapsulate *chunks*. A chunk is a unit of data allocation. Each chunk provides a server interface exported by the message it belongs to. As an example, messages that need to be causally ordered have a chunk that provides a `Causal` server interface. This interface defines methods to set and get a matrix clock. A message may encapsulate other messages and is uniquely identified by an interface called `Message` that gives access to the message's chunks and encapsulated messages.

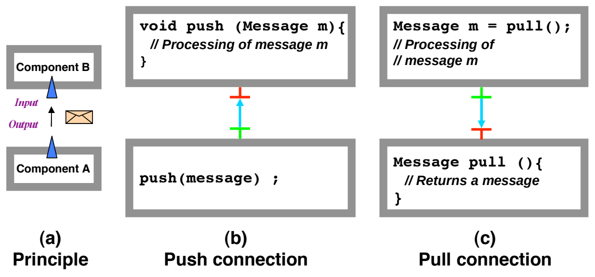

Figure 5. Connection between input/output interfaces

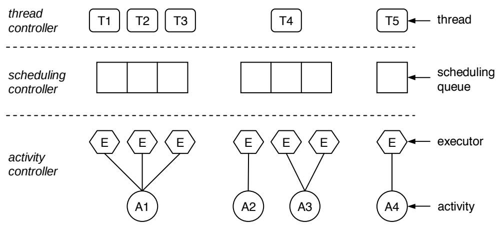

Figure 6. Activity management

Activity management A Dream component can either be passive or active. An active component has its own *activities*; a passive component doesn't, i.e. calls to other component interfaces can only be made in the activity of a calling component. An activity is a Java object implementing a method `run`. This method is executed as long as it returns a positive integer.

Active components have three controllers depicted in Figure 6, which we now describe. The **activity controller** allows the component to register, unregister, start, and stop activities. Activities are wrapped by *executors* that are in charge of the lifecycle of the activities they wrap. In particular, when a component needs to be stopped, the executors guarantee a safe interruption of the activies of the component. The number of executors wrapping a given activity is specified as a parameter of the activity's registration. Executors are executed by threads managed by a **thread controller**. Each thread is associated to a *scheduling queue*, where the **scheduling controller** places the next activities to be executed. This architecture allows fine-grained control over threads executing in the system, which is a required feature to build scalable asynchronous middleware as illustrated by the SEDA framework [38].

The Dream *library and tools* By lack of space we only describe the core components of the Dream library, i.e. the components encapsulating functions and behaviors commonly found in an asynchronous middleware. Note that the library also contains specific components developed

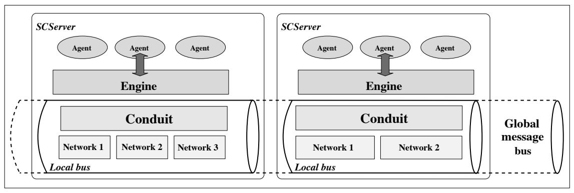

Figure 7. Two interconnected agent servers

for particular middleware: for instance, components implementing event-reaction processing. Examples of such components are given with the example presented in the next section.

Message queues are used to store messages. Queues differ by the way messages are sorted (FIFO, LIFO, causal order, etc.), and the behavior of the queue when the capacity is exceeded (blocks vs. removes messages), when the queue is empty, etc.

Transformers have one input to receive messages and one output to deliver transformed messages. Typical transformers include stampers.

Routers have one input and several outputs (also called "routes"), and route messages received on their input to one or several routes.

Filters have one input and one output. Messages received on the input are either delivered on the output, or deleted.

Aggregators have one or several inputs to receive the messages to be aggregated, and one output to deliver the aggregated message.

De-aggregators implement aggregators' reverse behavior, i.e. they take an aggregated message and generate appropriate individual messages from it.

Channels allow message exchanges between different address spaces. Channels are distributed composite components that encapsulate, at least, two components: a *ChannelOut* — which aims at sending messages to another address space —, and a *ChannelIn* — which can receive messages sent by the ChannelOut.

4.3.2. Re-engineering JORAM

This section presents how DREAM has been used to re-engineer JORAM. We first briefly present JORAM. Then we detail its implementation using DREAM. Finally, we compare both implementations in terms of configurability and performance.

A brief introduction to JORAM JORAM comprises two parts: the ScalAgent message-oriented middleware (MOM) [17], and a software layer on top of it to support the JMS API.

The ScalAgent MOM is a fault-tolerant platform, written in Java, that combines asynchronous message communication with a distributed programming model based on autonomous software entities called *agents*. Agents behave according to an "event → reaction" model. They are persistent and each reaction is instantiated as a transaction, allowing recovery

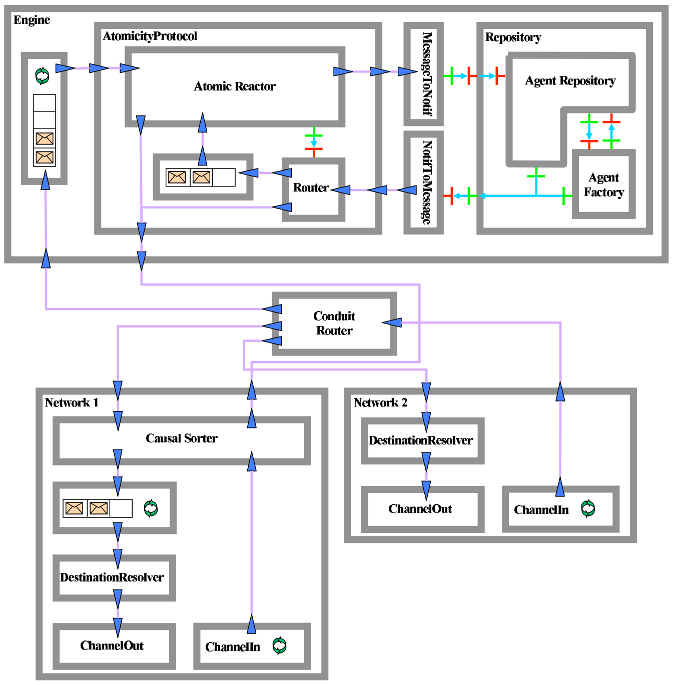

Figure 8. Architecture of an agent server

in case of node failure. The ScalAgent MOM comprises a set of agent servers. Each agent server is made up of three entities. The *Engine* is responsible for the creation and execution of agents; it ensures their persistency and atomic reaction. The *Conduit* routes messages from the engine to the networks. The *Networks* ensure reliable message delivery and a causal ordering of messages between servers.

Implementing JORAM *using* DREAM We have implemented the ScalAgent MOM using DREAM (see Figure 8). Its main structures (networks, engine and conduit) have been preserved to facilitate the functional comparison between the ScalAgent MOM and its DREAM re-implementation.

The **engine** comprises two main components: the `AtomicityProtocol` composite that ensures the atomic execution of agents; the `Repository` composite, which is in charge of creating and executing agents. Two typical **networks** are depicted. Both are composite components encapsulating a `TCPChannelIn`, a `TCPChannelOut` and a `DestinationResolver` component. The latter is a *transformer* that adds the information required by the `TCPChannelOut` component (i.e. IP address, and port number). The `Network 2` composite contains two more components: the `CausalSorter` causally orders messages; the message queue decouples the workflows of the engine and the network. The **conduit** is implemented by a router.

4.3.3. Configurability assessment

A first benefit of the Dream implementation comes from the ability to easily change provided non-functional properties. For instance, it is straightforward to remove causal ordering, or to remove the atomic protocol ensuring transactional execution of agents. Both modifications can be programmatically done at runtime. On the other hand, removing these properties from the ScalAgent MOM requires modifying and recompiling its source code. Moreover, by implementing the conduit as a router, an agent server can have multiple engines, which is not the case in the ScalAgent implementation. This is interesting for two reasons: it allows the parallelization of agent executions (within an agent server, agent executions are serialized [17]) and different non-functional properties can be simultaneously enforced (persistency, atomicity).

Another benefit brought by implementing the MOM with Dream is that it is easy to change the number of active components encapsulated within the agent server. The architecture we have presented in Figure 8 involves three active components for an agent server with one network. A mono-threaded architecture can be obtained by removing the message queues encapsulated within the engine and the network.

A last experiment we have done, is to build an agent server for mobile equipments. These equipments may be temporarily disconnected from the network and have limited storage capacity. To overcome these limitations, we have built an engine whose message queue is replaced by a `TCPChannelIn` component, and which encapsulates a `TCPChannelOut` component to send messages. Another device acts as a proxy and message storage unit for this engine. This architecture preserves the MOM functionnality, while saving memory: it is mono-threaded; messages are pulled instead of pushed; it has no `CausalSorter` and `DestinationResolver` components.

4.3.4. Performance comparisons

Measurements have been performed to compare the efficiency of the same application running on the ScalAgent MOM and on its Dream implementation. The application involves four agent servers; each one hosts one agent. Agents in the application are organized in a virtual ring. One agent is an initiator of rounds. Each round consists in forwarding the message originated by the initiator around the ring. We did two series of tests: messages without payload and messages embedding a 1kB payload. Experiments have been done on four PC Bi-Xeon 1,8 GHz with 1Go, connected by a Gigabit Ethernet adapter, running Linux kernel 2.4.20.

Table II shows the average number of rounds per second, and the memory footprint. We have compared two implementations using Dream with the ScalAgent implementation. The first implementation using Dream is not dynamically reconfigurable. As we can see, the number of rounds is slightly better ($\approx$ 1,2 to 2%) than in the ScalAgent implementation. Concerning the memory footprint, the Dream implementation requires 9% more memory, which can be explained by some of the structure needed by Fractal ($\approx$ 70kB) and the fact that each component has several controller objects. This memory overhead is not significant for standard

MOM	Number of rounds 0 KB	1 KB	Memory footprint (KB)
ScalAgent	325	255	4 × 1447
DREAM (non-reconf.)	329	260	4 × 1580
DREAM (reconf.)	318	250	4 × 1587

Table II. Performance of DREAM implementations vs ScalAgent implementation

MOM	Number of rounds 0 kB	1 kB	Memory footprint (kB)
Dream (3 threads)	329	260	4 × 1580
Dream (2 threads)	346	268	4 × 1516
Dream (1 thread)	370	279	4 × 1452

Table III. Impact of the concurrency level

MOM	Number of rounds 0 kB	1 kB	Memory footprint (kB)
ScalAgent	182	150	4 × 1447
Dream (4 agent servers)	188	153	4 × 1580
Dream (2 agent servers)	222	181	2 × 1687
Dream (1 agent server)	6597	6445	1 × 1900

Table IV. Impact of the number of engines by agent server

PC. The second implementation is dynamically reconfigurable (in particular, each composite component supports a life-cycle controller and a content controller). This implementation is slower than the ScalAgent one ($\approx$ 2,2 to 2%) and only requires 7kB more than the non-reconfigurable implementation made using DREAM.

Table III reports on experiments we have done to assess the impact of the concurrency level on the performances of the ScalAgent MOM. We compare three architectures built using DREAM that differ by the number of active components they involve. In the 2-thread architecture the message queue encapsulated in the network has been removed. In the mono-threaded architecture, both active message queues have been removed (Engine and Network). We see that, in this particular case, reducing the number of active components improves the number of rounds (+ 5 to 3% for the 2-thread architecture, and + 12 to 7% for the mono-threaded architecture). This can be explained by the fact that agents are organized in a virtual ring, thus each agent server only processes one message at a time. As a consequence, only one thread is necessary.

We have also evaluated the gain brought by changing the configuration in a multi-engine agent server. We have compared four different architectures: the ScalAgent one, an equivalent

Dream configuration with four mono-engine agent servers, a Dream configuration with two 2-engine agent servers, and a Dream configuration with one 4-engine agent server. Contrary to the previous experiment, agent servers are hosted by the same PC. Moreover, in the latter case, agents are placed so that two consecutive agents in the virtual ring are hosted by different agent servers. Table IV shows that using two 2-engine agent servers improves the number of rounds by 18% and reduces the memory footprint by 47%. The increase of the number of rounds can be explained by the fact that matrix clocks used by the causal sorter have a n^2 size, n being the number of agent servers. Thus, limiting the number of agent servers reduces the size of the matrix to be sent with messages, and tested before delivering them. Table IV also shows that using a 4-engine agent servers is 29 (35 for 1kB messages) times faster than using four mono-engine agent servers. This result may seem surprising, but can be easily explained by the fact that inter agent communication do not transit via the network components. Instead, the router directly sends the message to the appropriate engine.

5. Related work

Component models The Fractal model occupies an original position in the vast amount of work dealing with component-based programming and software architecture [36, 27, 25], because of its combination of features: hierarchical components with sharing, support for arbitrary binding semantics between components, components with selective reflection. Aside from the fact that sharing is rarely present in component models (an exception is [28]), most component models provide little support for reflection (apart from elementary introspection, as exemplified by the second level of control in the Fractal model discussed in Section 2). A component model that provides extensive reflection capabilities is OpenCOM [20]. Unlike Fractal, however, OpenCOM defines a fixed meta-object protocol for components (in Fractal terms, each OpenCOM component comes equipped with a fixed and predetermined set of controller objects). With respect to industrial standards such as EJB and CCM, Fractal constitutes a more flexible and open component model (with hierarchical composites and sharing) which does not embed predetermined non functional services. It is however perfectly possible to implement such services in Fractal, as demonstrated e.g. by the development of transactional controllers in [33]. Note also that Fractal is targeted at system engineering, for which EJB or CCM would be inadequate.

Software architecture in Java Several component models for Java have been devised in the last ten years. Apart from "standardized" models such as Java Beans, Enterprise Java Beans (EJB) or OSGI [4], we find open source initiatives such as Avalon [5] which is a general component model, Kilim [9], Pico [10] and Hivemind [7] which are targeted towards software configuration, Spring [12], Carbon [6], and Plexus [11] which are targeted towards component containers (in the line of EJB). These models suffer generally from the lack of extensibility and tailorability mentioned in the introduction. Carbon is probably the closest from Fractal as it provides extensibility and dynamicity through a mechanism based on *decorators* and *interceptors* and a JMX-based supervision. Two recent proposals for Java-based component programming include Jiazzi [26] and ArchJava [13]. Unlike these works, our approach to component-based programming in Java does not rely on language extensions for configuration purpose: JULIA is a small run-time library, complemented with simple byte-code generators. This, coupled with the reflective character of the FRACTAL model, provides for a more dynamic and extensible basis for component-based programming than Jiazzi, ArchJava, works cited above and most existing architecture description languages (ADLs). Note that FRACTAL and JULIA directly support arbitrary connector abstractions, through the notion of bindings. We have, for instance, implemented synchronous distributed bindings with an RMI-like semantics just by wrapping the communication subsystem of the Jonathan Java ORB [23], and asynchronous distributed bindings with message queuing and publish/subscribe semantics by similarly wrapping message channels from the DREAM library introduced in the previous section. ArchJava also supports arbitrary connector abstractions [14], but provides little support for component reflection as in FRACTAL and JULIA. Unlike JULIA, however, ArchJava supports sophisticated type checking that guarantees communication integrity (i.e. that components only communicate along declared connections between ports - in FRACTAL, that components only communicate along established bindings between interfaces).

Combining aspects and components The techniques used in JULIA to support the programming of controller and interceptor objects in a FRACTAL component membrane are related to several recent works on the aspectualization of components or component containers, such as e.g. [22, 30, 32, 8, 12]. The mixin and aspect code generators in JULIA provide a lightweight, flexible yet efficient means to aspectualize components. In line with its design goals, JULIA does not seek to provide extensive language support as AOP tools such as AspectJ or JAC provide. However such language support can certainly be build on top of JULIA. Prose [31] provides dynamic aspect weaving (whereas JULIA currently supports only load-time controller generation), with performance which appears to be comparable to that of JULIA. Prose, however, relies on a modified JVM, which makes it impractical for production use. In contrast, JULIA can make use of standard JVMs, including JVMs for constrained environments.

6. Conclusion

We have presented the FRACTAL component model and its Java implementation, JULIA. FRACTAL is *open* in the sense that FRACTAL components are endowed with an extensible set of reflective capabilities (controller and interceptor objects), ranging from no reflective feature at all (black boxes or plain objects) to user-defined controllers and interceptors, with arbitrary introspection and intercession capabilities. JULIA consists in a small run-time library, together with bytecode generators, that relies on mixins and load time aspect weaving to allow the creation and combination of controller and interceptor classes. We have evaluated the effectiveness of the model and its Java implementation, in particular through the re-engineering of an existing open source message-oriented middleware. The simple application benchmark we have used indicates that the performance of complex component-based systems built with JULIA compares favorably with standard Java implementations of functionally equivalent systems. In fact, as our performance evaluation shows, the gains in static and dynamic configurability can also provide significant gains in performance by adapting system configurations to the application context.

FRACTAL and JULIA have already been, and are being used for several developments, by the authors and others. We hope to benefit from these developments to further develop the FRACTAL component technology. Among the ongoing and future work we can mention: the development of a dynamic ADL, the exploitation of containment types and related type systems to enforce architectural integrity constraints such as communication integrity, the investigation of dynamic aspect weaving techniques to augment or complement the JULIA toolset, and the formal specification of the FRACTAL model with a view to assess its correctness and to connect it with formal verification tools.

Availability JULIA is freely available under an LGPL license at the following URL: `http://fractal.objectweb.org`.

REFERENCES

1. ASM: A Java Byte-Code Manipulation Framework, 2002. Objectweb, `http://www.objectweb.org/asm/`.
2. Java Message Service Specification Final Release 1.1, Mars 2002. Sun Microsystems, http://java.sun.com/products/jms/docs.html.
3. JORAM: Java Open Reliable Asynchronous Messaging, 2002. Objectweb, `http://joram.objectweb.org/`.
4. OSGi Service Platform, Release 3, 2003. `http://www.osgi.org/`.
5. The Apache Avalon project, 2004. `http://avalon.apache.org/`.
6. The Carbon project, 2004. `http://carbon.sourceforge.net/`.
7. The Hivemind project, 2004. `http://jakarta.apache.org/hivemind`.
8. The JBoss Aspect Oriented Programming project, 2004. `http://www.jboss.org/`.
9. The Kilim project, 2004. Objectweb, `http://kilim.objectweb.org/`.
10. The PicoContainer project, 2004. `http://www.picocontainer.org/`.
11. The Plexus project, 2004. `http://plexus.codehaus.org/`.
12. The Spring framework, 2004. `http://www.springframework.org/`.
13. J. Aldrich, C. Chambers, and D. Notkin. Architectural Reasoning in ArchJava. In *Proceedings 16th ECOOP*, 2002.
14. J. Aldrich, V. Sazawal, C. Chambers, and David Notkin. Language Support for Connector Abstractions. In *Proceedings 17th ECOOP*, 2003.
15. D. Ancona, G. Lagorio, and E. Zucca. A Smooth Extension of Java with Mixins. In *ECOOP'00, LNCS 1850*, 2000.
16. G. Banavar, T. Chandra, R. Strom, and D. Sturman. A Case for Message Oriented Middleware. In *Lecture Notes in Computer Science*, volume 1693, pages 1–18, Bratislava, Slovak Republic, September 1999. 13th International Symposium on Distributed Computing. ISBN 3-540-66531-5.
17. L. Bellissard, N. de Palma, A. Freyssinet, M. Herrmann, and S. Lacourte. An Agent Plateform for Reliable Asynchronous Distributed Programming. In *Symposium on Reliable Distributed Systems (SRDS'99)*, Lausanne, Switzerland, October 1999.
18. E. Bruneton, T. Coupaye, and J.B. Stefani. The Fractal Component Model. Technical report, Specification v2, ObjectWeb Consortium, 2003.
19. A. Carzaniga, D. Rosenblum, and A. Wolf. Design and Evaluation of a Wide-Area Event Notification Service. *ACM Transactions on Computer Systems*, 19(3):332–383, 2001.
20. M. Clarke, G. Blair, G. Coulson, and N. Parlavantzas. An Efficient Component Model for the Construction of Adaptive Middleware. In *Proceedings of the IFIP/ACM Middleware Conference*, 2001.
21. P. David and T. Ledoux. Towards a Framework for Self-adaptive Component-Based Applications. In *DAIS 2003, LNCS 2893*, 2003.
22. F. Duclos, J. Estublier, and P. Morat. Describing and Using Non Functional Aspects in Component Based Applications. In *AOSD02*, 2002.

23. B. Dumant, F. Dang Tran, F. Horn, and J.B. Stefani. Jonathan: an open distributed platform in Java. *Distributed Systems Engineering Journal, vol.6*, 1999.
24. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W Griswold. An Overview of AspectJ. In *ECOOP 2001, LNCS 2072*, 2001.
25. G. Leavens and M. Sitaraman (eds). *Foundations of Component-Based Systems.* Cambridge University Press, 2000.
26. S. McDirmid, . Flatt, and W.C. Hsieh. Jiazzi: New-age components for old-fashioned Java. In *Proceedings OOPSLA '01, ACM Press*, 2001.
27. N. Medvidovic and R. N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Trans. on Soft. Eng., vol. 26, no. 1*, 2000.
28. G. Outhred and J. Potter. A Model for Component Composition with Sharing. In *Proceedings ECOOP Workshop WCOP '98*, 1998.
29. R. Pawlak, L. Seinturier, L. Duchien, and G. Florin. JAC: A Flexible Solution for Aspect-Oriented Programming in Java. In *Reflection 2001, LNCS 2192*, 2001.
30. R. Pichler, K. Ostermann, and M. Mezini. On Aspectualizing Component Models. *Software – Practice and Experience*, 2003.
31. A. Popovici, G. Alonso, and T. Gross. Just in time aspects: Efficient dynamic weaving for Java. In *AOSD03*, 2003.
32. A. Popovici, G. Alonso, and T. Gross. Spontaneous Container Services. In *17th ECOOP*, 2003.
33. M. Prochazka. Jironde: A Flexible Framework for Making Components Transactional. In *DAIS 2003, LNCS 2893*, 2003.
34. N. Rivierre and T. Coupaye. Observing component behaviors with temporal logic. In *8th ECOOP Workshop on Correctness of Model-Based Software Composition (CMC '03)*, 2003.
35. Robert Strom, Guruduth Banavar, Tushar Chandra, Marc Kaplan, Kevan Miller, Bodhi Mukherjee, Daniel Sturman, and Michael Ward. Gryphon: An Information Flow Based Approach to Message Brokering. In *International Symposium on Software Reliability Engineering (ISSRE'98), fast abstract*, Paderborn, Germany, November 1998.
36. C. Szyperski. *Component Software, 2nd edition.* Addison-Wesley, 2002.
37. R. van Renesse, K. Birman, and W. Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Transactions on Computer Systems*, 21(2), 2003.
38. M. Welsh, D. Culler, and E. Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *Proceedings of the 18th Symposium on Operating Systems Principles (SOSP'01)*, Banff, Canada, 2001.

B Fractal Component-Based Software Engineering (The Fractal Component Model and Ecosystem)

Fractal Component-Based Software Engineering

Thierry Coupaye[1] and Jean-Bernard Stefani[2]

[1] France Telecom R&D
thierry.coupaye@orange-ftgroup.com
[2] INRIA
Jean-Bernard.Stefani@inrialpes.fr

Abstract. This article is a report on the 5th international workshop devoted to the Fractal component model that took place the 4th of July 2006 in Nantes, France, as an ECOOP workshop. Prior to that, the article provides some background on the Fractal project and previous Fractal workshops for readers who are not familiar with Fractal.

1 Introduction

We are witnessing a tremendous expansion in the use of software in scientific research, industry, administration and more and more in every day life. With the advent of the Internet and more globally the convergence between telecommunications and computing, software has become omnipresent, critical, complex. Time-to-market of services, which rely on system engineering (operating systems, distributed systems, middleware), is becoming a strategic factor in a competitive market in which operation (deployment, administration) costs are much higher than development costs.

In this context, component-based software architectures have naturally emerged as a central focus and reached momentum in different fields of computing because Component-Based Software Engineering (CBSE) is generally recognized as one of the best way to develop, deploy and administrate increasingly complex software with good properties in terms of flexibility, reliability, scalability[3] - not to mention lower development cost and faster time-to-market through software reuse and programmers productivity improvements.

Fractal is an advanced component model and associated on-growing programming and management support devised initially by France Telecom and INRIA since 2001. Most developments are framed by the Fractal project inside the ObjectWeb open source middleware consortium. The Fractal project targets the development of a reflective component technology for the construction of highly adaptable and reconfigurable distributed systems.

[3] As stated in the conclusions of the 7th International Symposium on CBSE (Edinburgh, Scotland, 2004): "Components are a way to impose design constraints that as structural invariants yields some useful properties".

2 The Fractal Ecosystem

2.1 Component Model

The Fractal component model relies on some classical concepts in CBSE: *components* are runtime entities that conforms to the model, *interfaces* are the only interaction points between components that express dependencies between components in terms of *required/client* and *provided/server* interfaces, *bindings* are communication channels between component interfaces that can be primitive, i.e. local to an address space or composite, i.e. made of components and bindings for distribution or security purposes.

Fractal also exhibits more original concepts. A component is the composition of a *membrane* and a *content*. The membrane exercices an *arbitrary reflexive control* over its content (including interception of messages, modification of message parameters, etc.). A membrane is composed of a set of *controllers* that may or may not export control interfaces accessible from outside the considered component. The model is *recursive (hierarchical) with sharing* at arbitrary levels. The recursion stops with base components that have an empty content. Base components encapsulate entities in an underlying programming language. A component can be shared by multiple enclosing components. Finally, the model is programming *language independent* and *open*: everything is optional and extensible[4] in the model, which only defines some "standard" API for controlling bindings between components, the hierarchical structure of a component system or the components life-cycle (creation, start, stop, etc).

The Fractal component model enforces a limited number of very structuring architectural principles. Components are runtime entities conformant to a model and do have to exist at runtime per se for management purposes. There is a clear separation between interfaces and implementations which allow for transparent modifications of implementations without changing the structure of the system. Bindings are programmatically controllable: bindings/dependencies are not "hidden in code" but systematically externalized so as to be manipulated by (external) programs. Fractal systems exhibits a recursive structure with composite components that can overlap, which naturally enforces encapsulation and easily models resource sharing. Components exercise arbitrary reflexive control over their content: each component is a management domain of its own. Altogether, these principles make Fractal systems self-similar (hence the name of the model): architecture is expressed homogeneously at arbitrary level of abstraction in terms of bindings an reflexive containment relationships.

2.2 Implementations

There exist currently 8 implementations (platforms)[5] providing support for Fractal components programming in 8 programming languages:

[4] This openness leads to the need for conformance levels and conformance test suites so as to compare distinct implementations of the model.

[5] Julia, AOKell, ProActive and THINK are available in the ObjectWeb code base. FracNet, FractTalk and Flone are available as open source on specific web sites.

- *Julia* was historically (2002) the first Fractal implementation[6], provided by France Telecom. Since its second version, Julia makes use of AOP-like techniques based on interceptors and controllers built as a composition of mixins. It comes with a library of mixins and interceptors mixed at loadtime (Julia relies very much on loadtime bytecode transformation as the main underlying technique thanks to the ASM Java bytecode Manipulation Framework). The design of Julia cared very much for performance: the goal was to prove that component-based systems were not doomed to be inefficient compared to plain Java. Julia allows for intra-components and inter-components optimizations which altogether exhibit very acceptable performance.
- *THINK* is a C implementation of Fractal, provided by France Telecom and INRIA Sardes, with a growing participation of STMicroelectronics and CEA, geared at operating and especially embedded systems development. Using THINK, OS architects can build OS kernels conforming to any kernel architecture: exo-kernel, micro-kernel... Minimal kernels can be built on bare hardware and basic functions such as scheduler and memory policies can be easily redefined or even not included. This helps achieve speed-ups and low memory footprints over standard general-purpose operating systems. THINK is also suggested for prototyping when using a complete OS would be a too heavy solution. It can also be used when implementing application-specific kernels, especially when targeting small platforms embedding micro-controllers. THINK comes along with KORTEX, a library of already existing system components, implementing various functions (memory management, schedulers, file systems, etc.) on various targets (e.g. ARM, PPC, x86).
- *ProActive* is a distributed and asynchronous implementation of Fractal targetting grid computing, developed by INRIA Oasis with a participation of France Telecom. It is a grid middleware for parallel, distributed, and concurrent computing, also featuring mobility and security in a uniform framework. It mixes the active object paradigm for concurrent programming (objects executing their own asynchronous activity in a thread) and the component paradigm for deployment and management.
- *AOKell* is a Java implementation by INRIA Jacquard and France Telecom similar to Julia, but based on standard AOP technologies (static weaving with AspectJ in AOKell v1 and loadtime weaving with Spoon in AOKell v2) instead of mixins. Also AOKell v2 is the first Fractal implementation that supports component-based membranes: Fractal component controllers can themselves be implemented as Fractal components. AOKell offers similar performance to Julia.
- *FractNet* is a .Net implementation of the Fractal component model developed by the LSR laboratory. It is essentially a port of AOKell on .Net, in which AspectDNG is used as an alternative aspect weaver to AspectJ or Spoon. FractNet provides for Fractal component programming in J#, C#, VB.Net and Cobol.Net languages.

[6] And sometimes considered for this reason as "the reference implementation" in Java.

– *Flone* is a Java implementation of the Fractal component model developed by INRIA Sardes for teaching purposes. Flone is not a full-fledge implementation of Fractal: it offers simplified APIs that globally reduce the openness and use of reflection of the general Fractal model so as to make teaching of component-based programming easier for students.
– *FracTalk* is an experimental SmallTalk implementation of the Fractal component model developed at Ecole des Mines de Douai. FracTalk focuses very much on dynamicity in component-based programming thanks the intrinsic dynamic nature of the SmallTalk language.
– *Plasma* is a C++ experimental implementation of Fractal developed at INRIA Sardes (with a participation of Microsoft Research) dedicated to the construction of self-adaptable multimedia applications.

2.3 Languages & Tools

A large number of R&D activities are being conducted inside the Fractal community around languages and tools, with the overall ambition to provide a complete environment covering the complete component-based software life cycle covering modelling, design, development, deployment and (self-)management. A representative but not exhaustive list of such activities is the following:

– development of formal foundations for the Fractal model, typically by means of calculi, essentially by INRIA Sardes,
– development of basic and higher levels (e.g. transactional) mechanisms for trusted dynamic reconfigurations, by France Telecom, INRIA Sardes and Ecole des Mines de Nantes (EMN),
– support for configuration, development of ADL support and associated tool chain, by INRIA Sardes, Jacquard, France Telecom, ST Micoelectronics,
– support for packaging and deployment, by INRIA Jacquard, Sardes Oasis, IMAG LSR laboratory, ENST Bretagne,
– development of navigation and management tools, by INRIA Jacquard and France Telecom,
– development of architectures that mix components and aspects (AOP), at the component (applicative) level and at the membrane (technical) level, by INRIA, France Telecom, ICS/Charles University Prague,
– development of specification models, languages and associated tools for static and dynamic checking of component behaviour, involving ICS/Charles University Prague, I3S/U. Nice, France Telecom, Valoria/U. Bretagne Sud,
– development of security architectures (access control, authentication, isolation), by France Telecom,
– development of QoS management architectures and mechanisms, for instance in THINK-based embedded systems, by France Telecom, or multimedia services with Plasma, by INRIA Sardes,
– development of semi-formal modelling and design methodologies (UML, MDA), models and tools, by CEA, Charles University Prague and others,
– ...

The most mature among these works are typically incorporated as new modules into the Fractal code base. Examples of such modules are the following:

- Fractal RMI is a set of Fractal components that provide a binding factory to create synchronous distributed bindings between Fractal components (la Java RMI). These components are based on a re-engineering process of the Jonathan framework.
- Fractal ADL (Architecture Description Languages) is a language for defining Fractal configurations (components assemblies) and an associated retargetable parsing tool with different back-ends for instantiating these configurations on different implementations (Julia, AOKell, THINK, etc.). Fractal ADL is a modular (XML modules defined by DTDs) and extensible language to describe components, interfaces, bindings, containment relationships, attributes and types - which is classical for an ADL - but also to describe implementations and especially membrane constructions that are specific to each Fractal implementation, deployment information, behaviour and QoS contracts or any other architectural concern. Fractal ADL can be considered as the favourite entry point to Fractal components programming (its offers a much higher level of abstraction than the bare Fractal APIs) that embeds concepts of the Fractal component model[7].
- FractalGUI is a graphical editor for Fractal component configurations which allows for component design with boxes and arrows. Fractal GUI can import/export Fractal configurations from/to Fractal ADL files.
- FScript is a scripting language used to describe architectural reconfigurations of Fractal components. FScript includes a special notation called FPath (loosely inspired by XPath) to query, i.e. navigate and select elements from Fractal architectures (components, interfaces...) according to some properties (e.g. which components are connected to this particular component? how many components are bound to this particular component?). FPath is used inside FScript to select the elements to reconfigure, but can be used by itself as a query language for Fractal.
- Fractal Explorer is a "graphical" (in fact a multi-textual windows system) management console that allows for navigation, introspection and reconfiguration of running Fractal systems in Java.
- Fractal JMX is a set of Fractal components that allows for automatic, declarative and non-intrusive exposition of Fractal components into JMX servers with filtering and renaming capabilities. Fractal JMX allows administrators to see a Fractal system as if it was a plain Java system instrumented "by hand" for JMX management: Fractal components are mapped to MBeans that are accessible by program or with a JMX console through a JMX server.

[7] It is worth noticing that Fractal ADL is not (yet) a complete component-oriented language (in the Turing sense), hence the need for execution support in host programming languages a.k.a. "implementations".

2.4 Component Library & Real Life Usage

Fractal has essentially been used so far to build middleware and operating system components. The current library of components engineered with Fractal that are currently available inside ObjectWeb include:

- DREAM, a framework (i.e. a set of components) for building different types (group communications, message passing, event-reaction, publish-subscribe) of asynchronous communication systems (management of messages, queues, channels, protocols, multiplexers, routers, etc.)
- GOTM, a framework for building transaction management systems (management of transactions demarcation, distributed commit, concurrency, recovery, resources/contexts, etc.)
- Perseus, a framework for building persistence management systems (management of persistency, caching, concurrency, logging, pools, etc.),
- Speedo, an implementation of the JDO (Java Data Object) standard for persistence of Java objects. Speedo embeds Perseus,
- CLIF, a framework for performance testing, load injection and monitoring (management of blades, probes, injectors, data aggregators, etc.)
- JOnAS, a J2EE compliant application server. JOnAS embeds Speedo (hence Perseus, Fractal, Julia, ASM),
- Petals, an implementation of Java Business Integration (JBI) platform, i.e. an Enterprise Software Bus.

Some of these components that embed Fractal technology are used operationally, for instance JOnAS, Speedo and CLIF by France Telecom: JOnAS is widely used by France Telecom[8] for its service platforms, information systems and networks by more than 100 applications including vocal services including VoIP, enterprise web portals, phone directories, clients management, billing management, salesman management, lines and incidents management.

3 Organization of the Workshop

3.1 History of Fractal workshops

The Fractal CBSE workshop at ECOOP 2006 was the 5th in the series[9].

The first workshop was held in January 2003 as an associated event of an ObjectWeb architecture meeting. The attendance was of about 35 people. 15 talks were given, organized in 5 sessions. The first session was a feedback session about the use of Fractal in Jonathan (a flexible ORB), JORAM (a JMS-compliant MOM) and ProActive (a distributed computing environment based on active objects). The second session was dedicated to Fractal implementation,s

[8] See http://jonas.objectweb.org/success.html for a more comprehensive list of operational usage of JOnAS.

[9] All programs and talks from Fractal CBSE workshops are available on the Fractal project web site at http://fractal.objectweb.org.

namely Julia and THINK. The third sessions was devoted to configuration tools, namely Kilim and Fractal GUI. The fourth session was dedicated to management and deployment, especially JMX management with Fractal JMX and connection with J2EE management and OSGi. The last session presented a conceptual comparison of Fractal and other component models.

The second workshop was held in March 2004 as an associated event of an ObjectWeb architecture meeting and ITEA Osmose project meeting. The attendance was of about 30 people. 10 talks were given, organized in 3 sessions. The first session was dedicated to tutorials on the Fractal model and Java tools (Fractal ADL, Fractal GUI, Fractal Explorer). The second session was dedicated to feedback from practical usage of Fractal in the Dream communication framework, the CLIF framework for load injection and performance evaluation and the GoTM open transaction monitor. The third session was dedicated to work in progress: components for grid computing with Fractal and ProActive, components and aspects, convergence of the Fractal and SOFA component models.

The third workshop was held in June 2005, again as an associated event of an ObjectWeb architecture meeting. The attendance was of about 20 people. It was mostly dedicated to discussions about components and aspects around AOKell (aspect-oriented programming of Fractal component membranes), FAC (Fractal Aspect Components: reification of aspects as components), and ”micro-controllers”. Another talk was given about the development of a formal and dynamic ADL.

The fourth workshop was held in November 2005 as a satellite of the ACM /IFIP/USENIX Middleware conference. The attendance was of more than 50 people. 8 talks about work in progress were given, framed by an introduction to Fractal and the Fractal project, and a final discussion about the evolution of the Fractal project. The technical talks described the recent developments concerning the Fractal ADL tool chain, the Fractal RMI ORB, the AOKell and ProActive implementations, reliability of Fractal components thanks to contracts (ConFract), behaviour protocols and model checking, with an original talk from the Nokia research center about dynamic and automatic configuration of components.

3.2 Call for proposals

The call for proposals, that was publicized on several mailing-lists (ObjectWeb, DBWorld, seworld, ACM SIGOPS France...), contained:

- a description and rationale for component-based architecture and its interest for the ECOOP conference;
- the expected audience: the Fractal community inside the ObjectWeb community hopefully enlarged thanks to ECOOP;
- the definition of scope of expected proposals: implementation and conformance test suites, model extensions, languages and tools, practical usage and feedback;
- and finally a description of the submission and selection process.

The submission and selection processes were rather light. Submissions were asked to contain 2 to 4 pages describing the work to be presented during the workshop. No full-length articles were asked for submission[10].

3.3 Selection and call for participation

More than 20 propositions were received, evaluated and discussed by the workshop organisers. Among them, 11 were selected for regular talks during the workshop. The selection was based on several individual criteria (technical maturity, originality, novelty) and also globally so as to cover a wide spectrum of activities around the Fractal component model and to make an interesting program with potential vivid discussions among participants. Most other proposals were very relevant but unfortunately could not fit in a one-day workshop, and were proposed to give place to poster presentations during breaks and lunch.

The final call for participation repeated the general items of the call for proposals and gave the detailed program with the list of regular talks and posters.

4 Tenue of the Workshop

The workshop took place the 3rd of July 2006. It was organized around 11 talks (typically 20 mn talk + 10 mn discussion) grouped in 5 sessions: Implementation and Basic Tools, Higher Languages and Tools, UML and MDA Design, Verification and Predictable Assembly, and Applications. 3 poster sessions also took place during coffee breaks and lunch. A final free discussion involving the around 30 participants closed the workshop.

4.1 Presentations and Discussions

The first morning session was devoted to implementations and basic tools for Fractal component programming.

L. Seinturier presented a joint work between INRIA Jacquard (L. Seinturier, N. Pessemier) and IMAG LSR laboratory (D. Donsez, C. Escoffier) towards a reference model for implementing the Fractal specifications in Java and the .Net platform. This preparatory work, fuelled by the development of the AOKell Fractal implementation and its port on the .Net platform, and a comparative analysis of the Julia implementation, advocates for a greater interoperability between Fractal implementations. The purpose of a Fractal implementation is to support the Fractal APIs and to offer mechanisms to compose control aspects inside membranes. Of course, all Fractal implementations support the Fractal API (with possible different conformance levels however) but offer generally different and incompatible mechanisms for building membranes. The aim of this line of work is to define some "Service Provider Interfaces" (SPI) that would

[10] A post-workshop editing and publishing activity to produce post-workshops proceedings was planned however.

embody programming conventions; implementations should follow these conventions so as to build assembly of, for instance, Julia and OAKell components and hopefully mix controllers/interceptors from different implementations. This line of work was acknowledged by the audience as very useful and important, and probably strongly connected to necessary efforts towards the definition of compliance test suites and benchmarks for Fractal implementations.

E. Özcan presented a status of the work in progress around THINK by STMicrolectronics (E. Özcan, M. Leclerc), France Telecom (J. Polakovic) and INRIA Sardes (J.-B. Stefani). The talk focused on recent developments of the ADL toolchain for THINK (Fractal ADL Factory) so as to make it more modular (finer-grained), extensible and retargetable, i.e. able to consider different back-ends corresponding to different hardware platforms. The talk concluded by listing other recent R&D activities and additions to the Kortex component library such as support for multi-processor platforms and support for customizable multimedia applications. The following discussion was not so much technical but concerned the collaborative management of the THINK code base. The THINK code base was historically managed by a few individuals from France Telecom and INRIA, with a quite clear direction and minimal collaborative decision making. Now, the growing implication of STMicrolectronics and others raises the question of how to choose between alternative propositions, e.g. concerning the design of the ADL tool chain for THINK, who is authorized to commit in the code base, who is authorized to create branches, etc.

The second session was devoted to higher languages and tools.

P.-C. David presented the work he did on FScript with T. Ledoux at Ecole des Mines de Nantes and France Telecom. FScript is a scripting language that allows for expressing reconfigurations of Fractal systems much more concisely, thanks to a higher level of abstraction than the bare Fractal APIs. FScript also includes FPath, a sublanguage/subsystem for navigation/query in Fractal architectures. It only comes with a Java backend for the time being but works are ongoing, e.g. at France Telecom, to use FScript to express reconfigurations in the THINK platform. One focus of the talk was the ACID-like transactional properties of FScript that would allow for *safe* reconfigurations. The vivid discussion following the talk revealed that this important but complex matter would/should require more developments.

R. Rouvoy presented the work on attribute-oriented programming around Fraclet with N. Pessemier, R. Pawlack and P. Merle at INRIA Jacquard. Fraclet is an annotation framework for Fractal components in Java. The motivation for this work is that component programming can be considered as verbose - and hence time consuming - by developers because the components code has to respect some conventions and provide meta-information as required by the Fractal model. Fraclet is composed of a library of annotations and plugins to generate[11] automatically various artifacts required by the Fractal component model (a.k.a. callbacks). Annotations provide a way to describe the component meta-

[11] Fraclet and attribute-oriented programming in general takes its roots in generative programming and aspect-oriented programming.

information directly in the source code of the content Java class. Fraclet plugins generate either Fractal component glue (use of Fractal APIs) or FractalADL definitions. Two implementations of the Fraclet annotation framework exist: Fraclet XDoc and Fraclet Annotation. Fraclet XDoc uses the XDoclet generation engine to produce the various artifacts required by the Fractal component model. Fraclet Annotation uses the Spoon transformation tool to enhance the handwritten program code with the non-functional properties of the component model. The talk emphasised two benefits of the approach. First, a reduction in development time and in the size of the components code produced "by hand". Second, a better support for software deployment and evolution: the presence in components code of architecture/deployment concerns facilitates the co-evolution of business and architecture/deployment code. This second benefit appeared as arguable from an industrial point of view: mixing, within the same file, business and deployment concerns might not appear as such a pleasant idea for software administrators. Also, a massive use of annotations is quite questionable with respect to code analysis and dependability in general. Most participants to the workshops were rather programmers than industrials and appeared quite enthusiastic about annotations and Fraclet anyway!

The third session was devoted to component modelling and more specifically to UML and MDA design.

V. Mencl presented a study with M. Polak at Charles University, Prague. They used their comparative analysis of UML 2.0 components and Fractal components to discuss possible mappings of Fractal concepts in UML. They actually proposed one specific mapping and instrumented it as a plug-in for the Enterprise Architect platform which is able to generate the Fractal ADL component descriptions, Java interfaces and a *skeleton* of the actual Java code of components. In the after-talk discussion, some possible future extensions were mentioned such as to *reverse engineer* UML models from Fractal ADL descriptions or runtime capture and representation in UML of a running Fractal system.

F. Loiret presented a study with D. Servat at CEA/LIST and L. Seinturier at INRIA Jacquard about modelling real-time Fractal components. This early work includes the definition of a EMF (Eclipse Modelling Framework) meta-model of Fractal IDL and ADL description, as well as the development of an Eclipse plug-in for actual generation of Fractal components targetting the THINK platform. The perspectives that were discussed include an extension of the meta-model to describe components behaviour and a reverse engineering tool chain to extract behaviour from the code of components.

The general discussion at the end of this modelling session acknowledged that there is probably not a unique direct mapping between UML and Fractal, especially because of specificities of Fractal such as component sharing and reflection (components controllers and membranes). However, thanks to UML/MDA (meta)modelling capabilities, different UML (meta)models could be defined to tackle Fractal specificities. People/teams interested by this line of work inside the Fractal community were encouraged to discuss further and hopefully to converge towards a common meta model (or at least to assess if one such a common model would make sense).

In the afternoon, the fourth session was devoted to verification tools and predictable assembly.

J. Kofron presented the work on behaviour protocols by J. Adamek, T. Bures, P. Jesek, V. Mencl, P. Parizek and F. Plasil at Charles University, Prague. Behaviour protocols are basically a formalism that allows for the specification of the expected behaviour of components in terms of legal sequences of operation invocations on components interfaces. A static behaviour protocol checker has been developed in the context of the SOFA component models for several years by Charles University. Recently, behaviour protocols have been ported on the Fractal platform through a partnership between Charles University and France Telecom. The result is a static checker and a dynamic checker that include Java code analysis of primitive components with the JavaPathFinder (JPF) model checker.

E. Madelaine presented a case-study of verification of distributed components behaviour with L. Henrio and A. Cansado at INRIA Oasis/I3S/U. Nice. The case study application itself has been defined in a partnership between Charles University and France Telecom to experiment behaviour protocols (cf. previous paragraph). E. Madelaine and al. used this application to experiment with their own verification formalism, *parameterized networks* with their supporting verification platform Vercors. This formal approach allows for model-checking of components behaviour (typically deadlock and reachability checking). The work also mentioned the proposition of a new Fractal ADL module (defined in collaboration with Charles University) for attaching behaviour specification and associated verification tools in architecture descriptions.

D. Deveaux presented a work with P. Collet, respectively at Valoria/U. Bretagne Sud and I3S/U. Nice, on contract-based built-in testing. The approach leverages previous works on built-in testing of Java classes by Valoria and ConFract, a contracting system for Fractal by I3S and France Telecom. It proposes to instrument each component under test (CUT) with, for instance, ConFract contracts which embody the particular testing information of this component and a test controller that would generate a test bed component encapsulating (containing) each CUT. A prototype is currently under development. Some questions arose from the audience concerning the adherence to ConFract and if the approach was only applicable during the design phase or whether it would be used in a deployed system. D. deveaux explained that the system would exhibit low dependancy to the contracting system (alternative contract systems may be used instead of ConFract) and would not be limited to unit testing, but could also handle admission, integration and regression test thanks to the dynamic configuration management capabilities in Fractal.

The fifth and last session was devoted to applications in real life of the Fractal technology.

N. Rivierre presented the work around JMXPrism with T. Coupaye at France Telecom. JMXPrism is a mediation layer that stands between the systems to be managed through JMX and management consoles or applications. JMXPrism provides a unique access point (embedding a JMX server) for managers that allows for the definition and management of *logical views* on the managed systems. JMXPrism prevents managers to access directly the managed systems and allows for filtering, renaming, etc. JMXPrims is implemented in Fractal which makes it very dynamic, allowing views and other components of a JMXPrims server to be changed very easily. JMXPrism embeds Fractal JMX, which was released some time ago as open source in the Fractal code base, and which allows for a declarative and non-intrusive exposition of Fractal components in JMX. JMXPrism has been used inside France Telecom to build a toy autonomic prototype controlling the creation of threads correlated to memory consumption. It has also been used more operationally in a grid information system project in partnership with Fujitsu to provide an homogenous view of resources in cluster on which resource sharing control was exercised to arbitrate two concurrently running applications: a visio-conference application exhibiting real-time QoS constraints and a batch-oriented scientific computing application.

G. Huang presented the last work with L. Lan, J. Yang and H. Mei at Peking University, Beijing, China on next generation J2EE servers. The work advocates for a combined use of reflective (applicative) components as embodied in Fractal or the ABC tool chain from Peking University and reflective middleware (especially EJB container) in future J2EE servers. Experiments are been conducted in PKUAS, an J2EE-compliant J2EE application server developed at Peking University. The talk raises up the engaged collaboration between ObjectWeb and OrientWare[12], a Chinese open source middleware consortium, as a suitable context for this line of work.

4.2 Final Discussion

The open discussion session was launched by a short talk by D. Caromel from INRIA Oasis, who reported on the Grid Component Model (GCM). GCM is an component model dedicated to grid systems that is being defined by the IST CoreGrid[13] network of excellence (NoE) along with the IST STREP project GridCOMP which is in charge of implementing, tooling and experimenting GCM. Fractal is considered as the basis for GCM and also as the main candidate to emerge as the standard component model for grid computing, at least in Europe. The talk recalled for some changes in the Fractal APIs that would be suitable for grid environments and that were discussed in previous Fractal workshop (e.g. multicast interfaces) but, more importantly, advocates for a close synergy between ObjectWeb/Fractal and CoreGrid/GridCOMP, i.e. a support of Fractal in CoreGrid and symmetrically a commitment from the Fractal community. This point was largely acknowledged as an important matter for the visibility and future of Fractal.

[12] http://www.orientware.org

[13] http://www.coregrid.net/

The discussion on the expected synergy between ObjectWeb/Fractal and CoreGrid/GridCOMP raised up a more general discussion about the evolution of the Fractal project. Some time ago was announced an evolution towards "Fractal v3". Some points were discussed during the previous Fractal workshop (November 2005), namely: i) evolution of the Fractal model specification (e.g. removal of some semantic ambiguities, changes and additions required for grid computing), including evolution in the organisation of the work on the specification with editors, editing committee and contributors, ii) evolution in the management of the Fractal code bases (e.g. cartography/matrix of (in)compatibilities between implementations and tools, conformance test suites) and iii) evolution of the Fractal web site (e.g. bibliography, success stories) and more generally of the management of the Fractal community (e.g. more structured workshops with CFP, program committee, proceedings; working groups inside the Fractal project). Since then, some elements contributed to "Fractal v3" in a quite informal way e.g. reflections on interoperability between Fractal implementations (cf. work by Seinturier and al. in the previous section), organisation of the two last workshops as satellite events of Middleware and ECOOP conferences (including CFP, PC and hopefully post-proceeding for Fractal Workshop at ECOOP), additions to the Fractal code base(s) (e.g. AOKell, FScript) and web site. The discussion at ECOOP, as well as previous informal discussions in particular on the Fractal mailing-list, revealed that some people were perhaps expecting a quicker evolution. Again, after a lively discussion, the workshop organisers pointed out that there might have been a misunderstanding and that what was intended by "Fractal v3" does not boild down to just a evolution of the specification of the model itself but refers to a collective effort with implication of many individuals that are part of the Fractal community so as to tackle the different issues at stake (implementations engineering and interoperability, conformance test suites, tools, common code base for uses cases and demonstrators, management of commit in code bases, web site, etc.).

From the evolution of Fractal, the discussion then jumped to the standardisation of Fractal. Several participants advocated for a more volunteer approach of the Fractal community towards standardization organisms. A vivid discussion took place to assess which standard committees/organizations would be most appropriate (ISO, IUT, Sun JCP, OMG...). Some others pointed out that standardization in the middleware area is a huge effort and the return on this investment not always remunerating. Most participants agreed that the group or institution they represent would not have much resources for such activities anyway. The question of standardization activities around Fractal remains largely open.

Acknowledgments We would like to thank the ECOOP conference and workshop organizers for their assistance in the preparation of this workshop. We would like to thank the Fractal community for its vitality, for having proposed so many talks at this workshop even though this was the 3rd Fractal workshop in less than a year. Thanks to A. Lefebvre for his careful reading and comments.

C Large Scale Management Architecture

Towards a Flexible Middleware for Autonomous Integrated Management Applications

Mehdi Kessis, Pascal Déchamboux*, Claudia Roncancio**, Thierry Coupaye*, Alexandre Lefebvre**
**France Telecom, Research & Development, MAPS/AMS*
28 chemin du Vieux Chêne, 38243 Meylan CEDEX, France
{Firstname.Lastname}@francetelecom.com
***LSR-IMAG*
{Firstname.Lastname}@imag.fr

Abstract

Enterprise and global-scale systems today might have thousands to millions of geographically distributed nodes and this number will increase over time. Managing efficiently such scattered systems becomes increasingly complex and requires powerful management capabilities. Traditional solutions to manage and control them seem to have reached their limits. In recent years, integrated management systems and services as well as autonomic systems have raised much interest in distributed systems and software engineering. This paper discusses the architectural issues facing the design of large-scale distributed management systems. Then it suggests a new flexible and scalable integrated management middleware to handle management problems in large-scale networked and heterogeneous systems.

Key words:
Integrated management, autonomic, component-based architecture.

1. Introduction

In recent years there has been a considerable growth in the use of distributed systems (peer-to-peer networks, clusters, pervasive computing, sensor networks, etc). These systems are scattered in our companies, administrations, homes and even in our pockets. Typically, they consist of large numbers of heterogeneous computing devices connected by communication networks, using various operating systems, resources and services, and user applications running on them. The dependability of our societies on such systems is becoming more and more noticeable. However, they become larger and more complex, heterogeneous and scattered. Traditional solutions to manage and to control them seem to have reached their limits. The size and the complexity of distributed system make it hard or even impossible to manage each of their components. Today, enterprise networks may count tens to thousands of managed resources. Telecommunication operators may manage thousands to millions of resources. For instance, France Telecom is managing more than one million "Livebox" home gateways. Due to the increased cost and complexity of managing such infrastructures manually, distributed computing systems are moving towards more autonomous operation and management.

This paper discusses an integrated management middleware that deals with such complex environments. This middleware plays the role of management broker that can be used by administrators or by management applications (i.e., management processes). It offers them (a) a customized view of the system through resource monitoring functions and (b) resource control management functions. We believe that such an infrastructure may offer a high degree of autonomy through management applications, by automating actions on groups of resources. We note that, in order to be able to automate such processes, a rich information model as well as a powerful programmatic model are needed.

This paper overviews our ongoing research. We aim at investigating new approaches to handle scalability, autonomy and heterogeneity issues in system management. After discussing new management needs and challenges, this paper proposes a middleware architecture to deal with these issues. Our work focuses on the monitoring activity, bringing flexibility and adaptability to the proposed solution.

2. Distributed management challenges

Today, large-scale distributed systems management is facing several major challenges. In this study, we focus on four of these challenges: autonomy, scalability, heterogeneity, and administrative isolation.

2.1. Autonomy

Managing efficiently such scattered systems becomes increasingly complex and requires powerful management capabilities. Traditional solutions to manage and control them seem to have reached their limits. In recent years, integrated management systems and services, autonomic systems have raised much interest in distributed systems and software engineering [14]. An autonomic system is capable to repair, configure, heal and protect itself [14]. The emerging field of autonomic distributed computing addresses the challenge of how to design and build distributed computing systems that can manage, heal and optimise themselves. Distributed computing systems are moving towards increasingly autonomous operation and management, in which their interacting components can organise, regulate, repair and optimise themselves without human intervention. [32]. These systems are intended to tackle administration complexity that is out of reach of human administrators, for instance handling a large number of alarms and notifications. Besides, automating management may reduce cost and improve efficiency. To automate management, we need at least three key elements: (a) representation, observation and monitoring capabilities, (b) decision rules and mechanisms and (c) control mechanisms.

2.2. Scalability

Scalability is a major problem for large-scale distributed systems. There is no commonly accepted definition of it [9]. In this paper, we consider the following definition of scalability: *"A scalable system is one that maintains constant, or slowly degrading, overheads and performance as its size increases"* [8]. In the past years, centralised network management has shown inadequacy for efficient management of large heterogeneous networks. As a result, several distributed approaches have been proposed to overcome the problem [21]. This is a main concern of our work because an enterprise network is an order of magnitude less complex than the infrastructure of some service providers, who monitor thousands to millions of resources. There are two key aspects of scalability involved in system management: the size of networks and the number of users. Service providers create extreme demands on both aspects[1]. Management systems should accommodate large numbers of participating nodes and they should allow applications to monitor large numbers of managed resources. Grouping and distributing management operations may improve scalability of the management system.

[1] The international Engineering Consortium (http://www.iec.org), Performance Management of Next Generation Networks.

2.3 Heterogeneity

The information model is a key feature in any management system [2]. It offers a view of managed resources (network, services, applications, etc.) to management applications. Today's networks involve heterogeneous resources. A service failure can be related to network or to application failures. In order to rapidly identify causes of failures and to understand the behaviour of complex managed resources, it is important to be able to describe heterogeneous managed resources and their interactions in the same way. The main object of integrated management [12] is to integrate different types of management (policy, user, network, services) in a unique infrastructure. Such infrastructure offers a complete view of the managed environment. To do so, we need a common description and representation of managed resources and their interactions.

2.4. Administrative Isolation

Traditionally, in large-scale networked systems, elements are grouped in managed domains. *"A domain is a set of objects to which a common management policy applies"* [18]. Each management domain is a logical partition of managed resources and management services. The set of managed objects may include computers, people, privileges, software processes, etc, depending on the purpose for which the domain is defined. In order to deal with large-scale networked and interconnected systems, domain management tools are needed. Such tools offer to the administrator the possibility to create, extend or merge new logical domains from existing primitive domains. Let us consider the France Telecom home gateway example. An example of primitive management domain could be the set of gateways related to a particular DSLAM. A management domain might contain the logical partition of gateways of a particular geographic zone, while another might contain the set of "LB1234" gateway model, in order to update their firmware. Administrator needs automated tools to build and to interact with management domains.

3. Towards a flexible and scalable integrated management middleware

3.1 A component-based model

Flexibility is a required property for managing large-scale networked systems. [11,7]. With a model such as the one proposed by Fractal [1], we believe that we can design, build and dynamically reconfigure component-based management infrastructure.

Fractal is a generic component model focusing on reconfiguration using flexible composition of components. It adopts a recursive view of components that may be nested. A component owns a *membrane* (i.e., a set of controllers), which realizes arbitrary forms of control over the content of the component. Composite components include other sub-components. A component sends and receives invocations through access points called interfaces. Such interactions require communication channels (named *bindings*) between some of the component interfaces. Figure 1 illustrates the architecture of a Fractal component. This composite component contains two sub-components and exposes control interfaces C_1 and C_2 as well as functional interfaces C_S and C_c.

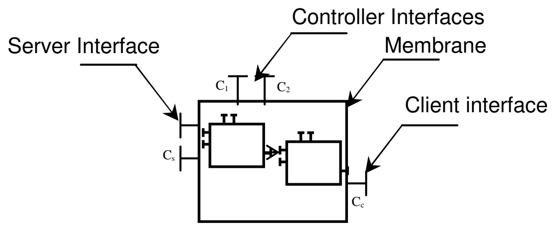

Figure 1 Example of Fractal components

Three main Fractal features are of particular interest:

a) *Component hierarchy*: composite components recursively contain components, ending with primitive components.

b) *Component sharing*: a (sub) component can be contained in several composite components. Typically, this feature can be used to model resources that are intrinsically shared.

c) *Components dynamicity:* bindings between components can be manipulated at runtime and is particularly interesting for management purpose. The model allows the definition of flexible bindings, since bindings may be themselves components.

Such properties are very interesting to design and build management domains. Figure 4 shows an example of mapping between domains and Fractal components. Disjoint domains are represented by disjoint components. Overlapping domains are represented by shared component. Hierarchical domains can be represented by composite domains.

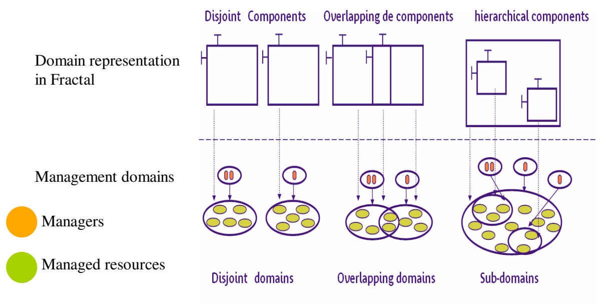

Figure 2 Management Domains in Fractal

3.2. Global management architecture

This section introduces the global architecture of an integrated management middleware that is positioned between management applications and managed resources. Managed resources are the set of physical resources (switches, PC, PDA, Set-Top Box, etc.) and logical resources (all or only a part of the OS, middleware, applications, services, etc.) available in an operator network.

A middleware relying on a flexible overlay network is a promising approach to overcome issues outlined in section 2. Such an approach allows management applications to construct an abstract view of the underlying network infrastructure and to federate different networks (IP, ad-hoc, etc). The middleware we propose builds an overlay network of mediation nodes that support management domains. The overlay network is functionally independent of the network infrastructure. It is formed by mediation nodes, which are interconnected through logical links. Figure 3 illustrates the global architecture of the proposed management infrastructure. Nodes collaborate in order to respond to queries of management applications. Each node represents one or many management domains. A node may contain one or many sub-domains (hierarchical relation). It interacts with several domains (links between nodes). This overlay builds an abstract representation of physical management domains (geographical for example) through management domains that correspond to specific management needs. (e.g., set of gateways model "LB1234"). Each node of the overlay offers a set of management services or tasks (information repository management, resource location service, query service, etc). As it can be noticed, these services cover only non functional management aspects delegated by management applications to the middleware.

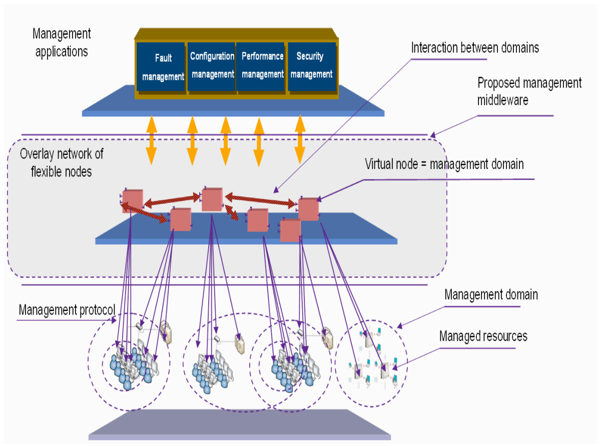

Figure 3 Global architecture of the management middleware

Both nodes structure and their interconnections are managed by the middleware administrator in a transparent way. The middleware hides management applications the complexity of the underlying infrastructure. The resulting overlay network is totally flexible, extensible and dynamically reconfigurable. Nodes and interconnections are dynamically reconfigurable.

To better understand the behaviour of these nodes, let us zoom inside one of them as depicted in Figure 4. Inside a node, a set of components work together in order to achieve a function. In Figure 4, the node offers 4 services: (i) events management service (that handles events and routes them to interested entities), (ii) a CIM repository (representation of the managed infrastructure), (iii) a query management service (for querying CIM repositories) and (iv) a repository for naming resources.

We believe that management applications requires rich information modelling to take into consideration physical and logical interdependent resources. Common Information Model (CIM) [4] is a standard for defining device, network and application characteristics so that system and network administrators and management programs can control heterogeneous devices and applications.

CIM also allows for vendor extensions. The adoption of such a model is key to overall interoperability for information storage and for system management environment.

The proposed middleware is based on two API:

a) *A configuration and deployment API*: It concerns the mediation nodes and management services. Network administrator, can build, deploy and configure the management middleware. After configuration and deployment, the middleware is ready to be requested by management applications.

b) *A mediation API*: This API is used by management applications. It permits them to communicate, indirectly, with managed resources, through our middleware.

Although both interfaces have different concerns, they are both managed by administrators at different level. Furthermore, the proposed middleware should use itself for its own management issues.

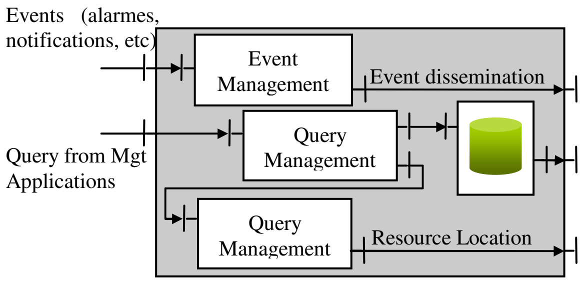

Figure 4 Internal structure of mediation node

4. Discussion

The choice of component-based architectures for managing networks and services have several advantages. V. Wadel et al. [17] consider them when designing management solutions within the world of telecommunication (flexibility, modularity, clear design, etc.).

The management applications are designed in order to be independent from the size of the network or from the number of properties to ensure. The management middleware aims at routing the requests over management information, supporting persistence of this information when necessary, locating resources, etc. It provides mediation nodes whose one of the primary roles is to ensure that its functions whatever are the conditions of the infrastructure that support the management network (i.e., an overlay network). The objective is that this middleware can adapt to such diverse situations as sensors network or as computation grids. This means that flexibility and adaptability are among the main challenges we target. These properties should ensure that our middleware can adapt to the various functional requirements of management applications, and can also adapt its own behaviour to the resources dedicated to its operation. We argue that component-based architecture is a major enabler towards this goal.

The Fractal component model we rely on allows the definition of components as assemblies of components and provides total control over the component used as well as the bindings between them. Hence, at the very end, the overlay can be considered a component composed of other components (e.g., the mediation nodes), managing them and their relationships as well. The overlay can then be configured and reconfigured in order to respond to management needs.

Autonomous management is seen as the only means to deal with large-scale management. This approach will lead to very complex applications to support the processes composing this autonomous management environment.

The information model on which the management applications rely is highly distributed by nature. So should be the middleware for supporting them. We consider that good work has been done for specifying the information model, especially with CIM [4]. Our objective is to be able to organise the management of this information space in a distributed manner while ensuring its safety, scalability, correctness (i.e., in a sense considering the implementation of a reliable distributed CIMOM). We also consider much simpler interfaces to manipulate this information model, especially within our Java implementation context. For example, we consider pure Java objects accessible through technologies such as EJB[2] or JDO[3] for giving access to the CIM repository.

5. Related Works

Several works studied large-scale systems and network management, during these two last decades [6, 10, 20, 8].

CIM/WBEM [6], a DMTF[4] standard, proposes a web-based management architecture. WBEM is built around the CIM model. The main WBEM architecture is composed of three main elements (management applications as client, WBEM server, WBEM providers (probes and actuators)). This architecture follows a flat and centralized model (Manager/Agent model). There is no M to M (Manager to Manager) communication. WBEM servers can be used to manage enterprise environments. However, it does not scale to large distributed environments. The administrative isolation is implemented through the concept of namespace (logical view of CIM instances and classes).

Yalagandula et al proposed SDIMS (Scalable Distributed Information Management System) [5]. It consists of a building block for large-scale distributed services. SDIMS aggregates information about large-scale networked systems and provides detailed views of information (and events) and summary views of global information. It ensures four properties: scalability, flexibility, administrative autonomy and robustness. This work concerns neither resource heterogeneity nor autonomic behavior.

Renesse et al [8] proposed a similar work: Astrolabe. This system gathers, disseminates and aggregates information about zones. A zone is recursively defined to be either a host or a set of non-overlapping zones. It supports scalability through hierarchy (zone hierarchy), flexibility through mobile code, robustness through a randomized peer-to-peer protocol and security through certificates. Each Astrolabe zone has a set of aggregation functions that calculates the attributes for the zone's MIB (SNMP like Management Information Base). Astrolabe is designed under the assumption that MIBs will be relatively small objects, a few hundred or even thousand bytes, not millions which limit its scalability.

Anerousis et al [10] proposed Marvel. This system is a distributed computing environment that allows the creation of scalable management services using intelligent agents and the world-wide web. Marvel builds on top of existing element management agents a hierarchy of servers that aggregate the underlying information in a synchronous or asynchronous fashion. Marvel is based on an information model that generates computed views of management information. These views follow an object-oriented model to store management information. Marvel requires that managed elements be organized into groups. Users can dynamically define these groups based on any factor that makes sense such as location or functionality. The object implementation of Marvel's views is proprietary and not extensible. Besides it does not address autonomy issue.

Recently, Bouchenak et al [15] proposed the JADE framework. It is an environment for implementing autonomic administration software. The main idea of this work consists on modelling the administrated system as a component based software architecture which provides means to configure the environment. A prototype of Jade was developed and used for deployment and fault management of clustered J2EE application. This work provides administrative isolation through composition and sharing relations. This work is based on an ad-hoc information model. The global vision of the proposed work in this paper is coherent and complementary with the autonomic management vision proposed in the European project IST Selfman [3], to which we actively participate.

6. Conclusion and Future works

In this paper we have studied large scale networked heterogeneous systems problem. We have identified four important properties that large-scale management

[2] http://java.sun.com/products/ejb/docs.html

[3] http://java.sun.com/products/jdo/

[4] Distributed Task Force Management; URL: http://www.dmtf.org

systems have to respect: autonomy, scalability, administrative isolation and heterogeneity. We suggest the architecture of a flexible middleware that respects these properties. The proposed middleware is based on the Fractal component model. It offers the administrator the possibility to build a scalable and dynamically reconfigurable overlay network of mediation nodes. Each node represents on or many management domains. The middleware offers the possibility to build, aggregate, select, and query management domains according to administrator needs. All these operations can be done without interrupting management applications activity. The different nodes cooperate to offer management applications several services (event filtering, aggregation, routing, storage management, etc). Scalability is achieved through the support of domain and through the distribution of the management activity.

We believe that the proposed middleware can be a powerful building block for autonomous management applications. A set of management policy can be defined for each management domain that we build. Automatic actions can be assigned to each of them. Scalability is achieved through distribution of the management infrastructure and grouping management operations. Heterogeneity is achieved through CIM local repositories, managed by the node of our overlay network. In these repositories, heterogeneous resources are described in a standard way. Administrative isolation is achieved through the different possible composition relations proposed by the Fractal component model. Autonomy is achieved through the reflexive aspect of the Fractal components.

We are studying the possibility to integrate a data steam management system to handle, in a scalable way, streams of events sent by probes and network equipments. This feature is particularly interesting for large event-based monitoring systems in real-time context. Besides, we are studying the possibility to make some of our node mobile. The ProActive[5] technology, based on the Fractal component model, has already experienced such an approach.

7. References

[1] E. Bruneton, T. Coupaye, and J.-B. Stefani. "Recursive and Dynamic Software Composition with Sharing". Proceedings of the Seventh International Workshop on Component-Oriented Programming (WCOP02), Malaga, Spain, June 10-14, 2002.

[2] J.-P. Martin-Flatin, "Toward Universal Information Models in Enterprise Management", in Proc. VLDB 2001 Workshop on Databases in Telecommunications (DBTel 2001), Rome, Italy, September 2001.

[3] P. Van Roy, A. Ghodsi, JB Stefani, S. Haridi, T. Coupaye, A. Reinefield, E. Winter and R. Yap. " Self management of large-scale distributed systems by combining structured overlay networks and components". Workshop IST NoE CoreGrid Integration, Greece, Nov 2005.

[4] Common Information Model Standard, URL: http://www.dmtf.org/standards/cim/

[5] P. Yalagandula and M. Dahlin, "A scalable distributed information management system ", Proceedings of the 2004 conference on Applications, technologies, architectures, and protocols for computer communications, session Distributed information systems, Pages: 379 – 390, Portland, Oregon, USA, 2004.

[6] Java Specification Request N°48 (JSR 48): WBEM Services Specification, URL: http://www.jcp.org/en/jsr/detail?id=48

[7] J. Won-Ki Hong, J. Kim and J. Park: "A CORBA-Based Quality-of-Service Management Framework for Distributed Multimedia Services and Applications". IEEE Network, Vol. 13, No. 2, (1999) 70-79

[8] R. V. Renesse, K. P. Birman, and W. Vogels, "Astrolabe: A Robust and Scalable Technology for Distributed System Monitoring, Management, and Data Mining", ACM Transactions on Computer Systems, Vol. 21, No. 2, May 2003, Pages 164–206.

[9] M. D. Hill, "What is scalability?". ACM SIGARCH Computer Architecture News, December 1990.

[10] N. Anerousis and G. Hjálmtysson, "View-based Management of Services in a Programmable Internetwork". Proc. of the 2000 Network Operations and Management Symposium, Honolulu, HI, April 2000.

[11] G. Goldszmidt, "Distributed Management by Delegation". 1996. Ph.D Thesis – Graduate, School of Arts and Sciences, Columbia University, New York.

[12] H.G. Hegering, S. Abeck and B. Neumair. "Integrated Management of Networked Systems: Concepts, Architectures, and their Operational Application". Morgan Kaufmann Publishers, 1999.

[14] J. O. Kephart and D. M. Chess." The vision of autonomic computing". IEEE Computer, 36(1):41–50, January 2003.

[15] S. Bouchenak, N. de Palma and D. Hagimont, "Autonomic administration of clustered J2EE applications". Proceedings of IFIP/IEEE International Workshop on Self-Managed Systems & Services (SelfMan 2005). 2005.

[16] M. Kahani, H.W. Peter Beadle, "Decentralized Approaches for Network Management", in SIGCOMM, July 1997.

[17] V. Wade, D. Lewis, C. Malbon, T. Richardson, L. Sorensen and C. Stathopoulos, "Component Integration Technologies for Telecoms Management Systems", TCD-CS, Technical Report, Trinity College Dublin Computer Science Department, 1999.

[18] M. Sloman, J-D. Moffett, "Domain model of autonomy". ACM SIGOPS European Workshop 1988

[5] http://www-sop.inria.fr/oasis/ProActive/

D Transactional Reconfiguration of Component-Based Architectures

Reliability of dynamic reconfigurations in component-based software systems

Marc Léger[1], Thierry Coupaye[1], and Thomas Ledoux[2]

[1] France Telecom R&D
28, chemin du Vieux Chêne
F-38243 Meylan
{marc.leger, thierry.coupaye}@orange-ftgroup.com
[2] OBASCO Group, EMN / INRIA, LINA
Ecole des Mines de Nantes
4, rue Alfred Kastler
F-44307 Nantes Cedex 3
thomas.ledoux@emn.fr

Abstract. This article is an analysis based on our experience with the Fractal component model of the need of reliability for dynamic reconfigurations in component based systems. We make a proposal to ensure this reliability, which can applied to concurrent reconfigurations. We started from the definition of ACID properties in the context of component models and we propose to use integrity constraints to define system consistency and transactions for guaranteeing the respect of these constraints at runtime. To deal with concurrency, we have to detect potential conflicts when composing reconfiguration operations.

1 Introduction

Dynamic reconfigurations in component-based software applications [MK96] are central to promissing approaches like autonomic computing [KC03]. There are many motivations to introduce modifications in a system at runtime: correction of security flaws or functional bugs, improvement of systems (e.g., performance optimizations), or adaptions to execution context changes.

Thanks to properties of component models like loosely coupling, reconfigurations can rely on component-based architectures [OMT98]. However, runtime modifications can let the system in an inconsistent state. From a structural point of view, the architecture of the system once reconfigured can be not in conformity with the component model or eventually system specific constraints (e.g. architectural invariants) anymore. From a functional point of view, a reconfiguration must not perturb the execution of the system (i.e., functional and non functional aspects need to be synchronized). Furthermore, in case of concurrent reconfigurations, reconfiguration must be synchronized between themselves.

In this paper, we focus to the reliability of runtime adaptations and we chose to base our work on the Fractal component model [BCL$^+$04] because of its support of dynamic and opened reconfigurations. In our approach, we tried to define each of the ACID properties [TGGL82] in the specific context of component-based systems an show how it can solve this reliability problem during adaptations. These properties are unifying concepts of transactions for distributed computation used essentially for supporting concurrency and recovery. We specify the consistency property by using integrity constraints about system structure and state. An example of a structural constraint at the level of component model is cycle-free component structure. Moreover we must avoid wrong execution flow of reconfiguration operations according to their semantics to ensure the isolation property.

This paper is organized as follows. Section 2 is an overview of dynamic reconfigurations in component models, with a focus on Fractal, and it shows what problems it raises regarding reliability. Then section 3 describes how transactions combined with integrity constraints can be a solution to these problems. Finally section 4 presents some related works before concluding in section 5.

2 The need of reliability for dynamic reconfigurations in component-based systems

2.1 Dynamic reconfigurations in component models

Dynamic reconfigurations allow modifications of a part of a system during its execution without stopping it entirely to keep the system partly available. Actually, maximization of the availability time is essential for some systems like entreprise application servers. Dynamic reconfigurations can involve every manageable element defined in the component model and reified at runtime, they can be:

– structural (e.g., addition or removal of elements like components, interfaces etc. and interconnection modifications with bind unbind operations),
– behavioral (e.g., lifecycle modification used to synchronize component activity with the rest of the system),
– linked to component deployment (e.g., component instantiation, destruction, migration),
– linked to component state (e.g., change of component attribute values),

Fractal [BCL+04] is a recursive component model with sharing and reflexive control. It is based on classic concepts of component (as a runtime entity), interface (an interaction point between components expressing provided and required services) and binding (a communication channel between component interfaces). A component consists of a membrane which can show and control a causaly connected representation of its encapsulated content. An Architecture Description Language (Fractal ADL [Fra]) is used to specify component configurations and there is notably a Java implementation of the model, Julia. Several controllers are defined to control bindings, the hierarchical structure, component lifecycle, attributes and names, but other controllers can be user-defined.

Operations in controllers constitute primitive reconfiguration operations and do either introspection or intercession (modifications) in the system. To compose operations, we consider sequences or parallel executions of intercession operations with conditions expressed by means of introspection operations in component configurations. An example of composite reconfiguration is component hotswap, a mechanism used to update a system where an old version of a component is replaced by a new one. In Fractal, this reconfiguration is composed of a sequence of several primitive reconfiguration operations, it implies to stop the component, unbind all its interfaces, remove it, add the new instantiated component, bind its interfaces and start it (a state transfert operation is used in case of stateful component).

2.2 The reliability problem with dynamic reconfiguring applications

A first problem when modifying a system at runtime is the synchronization between reconfigurations and the functionnal execution of the system. Actually, the part of the system which is modified could be unavailable for functional execution during the reconfiguration time. To take the hotswap example with a stateful component, calls on the old component must be blocked until a a "quiescent state" [KM90] is reached, then the state must be transfered, finally previous calls are forwarded towards the new component.

A second problem at the model level is about consistency violation by reconfigurations. First of all, we must make clear what exactly consistency is for component-based systems. Component models and application models should define what this consistent system is, especially in term of structure. For instance, we may want to add a structural constraint about the number of subcomponents of a composite component. In Fractal, the specification of the component model is not always sufficient and we want to express some integrity constraints on systems. So we must ensure the conformity of the system to the model and constraints after reconfigurations.

The third and last problem we identified is linked to the composition of reconfiguration operations. A prerequisite is the separation of concerns between the functional part and the control part of systems. Then separation between introspection operations and intercession operations must be explicit. Once these operations have been identified, the semantics of reconfiguration operations implies there can be some conflicts between them in case of compostion and for synchronization between several reconfigurations (e.g., in Fractal it is mandatory to unbind all component interfaces before removing the component from its super-component).

3 A transactional approach to ensure reliable reconfigurations

3.1 ACID properties in the context of dynamic reconfigurations

We think that well-defined transactions associated with structural and behavioral constraints verification is a means to guarantee the reliability of reconfigurations in component models, i.e. to solve problems we identified in the section

2.2. As any reconfiguration operation could lead the system to an inconsistent state, each reconfiguration must always be included in a transaction. In this context, we define the meaning of ACID properties as follows:

- **Atomicity**: either all happen or none happen, that is to say either the system is reconfigured or it is not. A reconfiguration transaction can be a single primitive reconfiguration operation or a more complex operation composed of several operations. Each reconfiguration operation must specify its reversible operation. Thus if a reconfiguration transaction goes badly and is rollbacked, it is possible to come back in a previous stable state by undoing operations. Transactions demarcation is either programmed in the language or automatic (a reconfiguration script corresponds to a transaction).
- **Consistency**: a transaction must be a correct transformation of the system state. So the reconfigured application must be conform to the component model and application specific constraints. That is to say consistency is given by integrity constraints essentially architectural invariants. A reconfiguration transaction can be commited only if the resulting system respects the constraints. Other faults like software and hardware failures (network and machines) are the responsibility of the commit protocol (e.g., 2 phase commit protocol).
- **Isolation**: several reconfiguration transactions are independant and any schedule of reconfiguration operations must be equivalent to their serialization. The scheduling must respect the operation semantics and conflicts. This property relies on the knowledge of the semantics of reconfiguration operations.
- **Durability**: once a reconfiguration completes with success (commit), the new state is persistent. For every transaction, operation are logged in a journal so that reconfigurations can be redone in case of failure. The application state (architecture and component state) is periodically checkpointed basically with ADL dumps and component state is saved in databases. So any component can be recovered in its last stable state resulting from the last successful reconfiguration. However, the only functional state we capture is the state which is well identified in the component model and is saved only at commit time of reconfigurations because we don't want to impose transactions at the functional level.

Only the first problem presented in 2.2 is not completely adressed by our approach because we do not fully modelise the functional execution flow of systems, we relies on the implementation of the component lifecycle operations with interceptors on component interfaces to realize the synchronization. A solution to the synchronization problem is to apply the hotswap protocol proposed in [KM90]. The guarantee we can bring is that the order of operations in the protocol is respected. Among the ACID properties we will especially focus in the following sections on two properties: consistency and isolation.

3.2 Integrity constraints to ensure system consistency

In our proposal, system consistency relies on integrity constraints and we want to express these constraints both at the application and at the model level. An integrity constraints is essentially a predicate which concerns the validity of an assembly of architectural elements but it can also concern component state. Examples of such constraints at the component model level are hierarchical integrity (bindings between components must respect the component hierarchy) or cycle-free structure (a component cannot contain itself to avoid infinite recursion). On the other hand, application specific constraints are used to specify invariants on a given system either on component types or directly on component instances designed by their names. Invariants can concern for example cardinality of sub-components in a super-component, two component interfaces which can never be unbound etc.

In an open world where reconfigurations are not anticipated at compile time, some component models like Fractal are relying on reflexive architectures to dynamically reconfigure systems by means of a runtime mapping between the system which is really executed and its model. So integrity constraints verified on the model will be also valid in the system. We represent the Fractal component model as a typed graph and then each fractal-based application is also a graph which is an instance of this typed graph. The instance graph is a more formal representation of the system provided at runtime by the reflexivity of the component model and is used to navigate in runtime applications. The vertexes are elements from the component model: components, functional interfaces, controllers, attributes and operations. The edges represent relations between the elements: composition links, binding links etc. Then the instance graph must always be well-typed regarding to the typed graph (i.e., conform to the component model) and the instance graph must respect integrity constraints. Therefore contraints at the model level can be specified on the typed graph and others on the instance graph. As the model is extensible and new user-defined controllers can be added, graphs should be also easily extensible in terms of elements and relations.

To express integrity constraints, we propose to use a DSL based on an extension of the query language in Fractal configurations FPath [DL06] to transform it into a real constraint language "à la OCL" [OCL05]. An advantage of the FPath language is that it can navigate both in the ADL and in the runtime system and it is already based on a graph representation of the system during execution. The constraint language must just have introspection capacity without side effects on the system. We want to express invariants, preconditions and postconditions in this language and we want notably to have quantifiers, collection operations and filters. The following basic example is a structural invariant constraint at the application level expressed in FPath (with its XPath 1.0 syntax like) where the component designed by the variable c can never be shared (it can only have one parent at the same time):

```
size(c/parent::*)=1
```

Constraints must be checked both at compile time on the component static configuration and at runtime. We consider checking constraints as far as possible before applying the reconfiguration on the system, eventually by code analysis of a dedicated reconfiguration language like FScript [DL06]. Constraints can also be checked either directly during the execution of the reconfiguration of the real system or by simulation on a local copy of the representation of the system (i.e., the instance graph) so as to limit the effect on the system in case of constraint violation.

3.3 Isolation of reconfigurations to support concurrency

We take the hypothesis that not only application components are distributed but also administrators. Furthermore, reconfiguration initiators are either humans (interactive reconfigurations) or the system itself (the system is able to auto-reconfigure). Concurrency in reconfigurations comes from the fact that one administrator can explicitely want to execute some operations in parallel, or several administrators can reconfigure the same system at the same time. The reconfiguration scheduler can also detect when it can launch parallel reconfiguration tasks to optimize the reconfiguration process.

As seen in section 2.1, reconfiguration operations are composable but all compositions are not valid. In Julia, operation semantics is hidden in controller implementations and so we want to make it explicit and we want eventually to be able to change it and to specify new primitive operations. So we need to express operation semantics in terms of preconditions and postconditions with our constraint language presented in section 3.2. We distinguish two types of conflicts between operations: parallel conflicts and execution dependencies. For two given reconfigurations *R1* and *R2* executed on the same system, a parallel conflict occurs if *R1* and *R2* modify the same manageable elements in the system model (e.g. bind and unbind operations). An execution dependency occurs if *R1* either need *R2* to be executed first (e.g. stop before unbind)or if *R1* cannot be executed after *R2*. That is to say *R2* postconditions cover or not *R1* preconditions.

```
// Example of a precondition for removing a component
operation: void removeSubComponent(Component sub);
preconditions :
// all interfaces of the sub-component are unbound (. is the current node)
not(exists(sub/interface::*[not(bound(.))]));
```

For concurrency management, we propose a pessimistic approach with locking. Our locking algorithm is based on operation semantics to avoid inconsistent operation compositions. We see two different possibilities for the locking algorithm. The first one is to lock directly reconfiguration operations. That is to say, either conflicts between operations are automatically calculated thanks to its preconditions and postconditions or it must define the operations with which it is in conflict. The second one is to use a modified DAG locking algorithm on our instance graph defined in 3.2. Then the lock granularity is defined by the manageable elements in the graph representation and for example a lock acquisition on a component also locks all its interfaces and every operations in each interfaces.

Another approach to locking is to constrain the execution order of reconfiguration operations. We propose to use a simple language inspired of behavior protocols in [PV02] to describe the desired execution order of reconfiguration operations, what we call behavioral reconfiguration constraints. The protocol compliance is checked at runtime by intercepting reconfiguration calls.

4 Related work

Many works on ADLs follow a static approach to check consistency of component-based architectures by compilation but only a few are interested in dynamic analysis of this consistency. We will focus here on other reflective component models which allow non anticipated (also called ad-hoc) reconfigurations.

FORMAware [MBC04] is relatively close to our work. This framework to program component-based application gives the possibility to constrain reconfigurations with architectural style rules. A transaction service manages the reconfiguration by stacking operations. The main difference with our proposal is our integrity constraints are more flexible than styles and they can be applied to every element of our component model. Moreover we define more formally reconfiguration operations to identify conflicts between them, our locking algorithm is then more precise than a simple lock on components and we consider introspection operations as reconfiguration operations.

Plastik [BJC05] is the integration of the OpenCOM component model and the ACME/Armani ADL. As in our solution, architectural invariants can be checked on ADL configuration or at runtime and constraints are expressed at the style level and at the instance level. However, reconfiguration cannot be generic composite reconfigurations with model elements in parameters and the execution, the operation semantics is not explicit and not extensible and the order of reconfiguration operation cannot be constrained as we can do with reconfiguration protocols.

5 Conclusion

Dynamic reconfiguration in component-based systems raises reliability problems, especially in open systems in which they are not anticipated. In this article, we identified the three following global problems based on our experience with the Fractal component model: synchronization between reconfiguration and the functional execution of systems, consistency regarding component and application models, and synchronization between reconfiguration operations. We focused more on the two last problems: the first one concerns conformity at runtime of systems with constraints and models, the second one deals with the validity of composition of reconfiguration operations.

We propose to use integrity constraints to define consistency for dynamic reconfigurations and to include these reconfigurations in transactions. We build a graph representation of our application at runtime thanks to the reflexivity of the Fractal component model and use a constraint language on this graph. Moreover we want to detect execution conflicts between reconfiguration operation in order to be able to compose them with reliability with eventually the specification of reconfiguration protocols. We are currently implementing this proposal in Julia, a Java implementation of the Fractal model.

References

[BCL+04] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. An open component model and its support in java. In Ivica Crnkovic, Judith A. Stafford, Heinz W. Schmidt, and Kurt C. Wallnau, editors, *CBSE*, volume 3054 of *Lecture Notes in Computer Science*, pages 7–22. Springer, 2004.

[BJC05] Thaís Vasconcelos Batista, Ackbar Joolia, and Geoff Coulson. Managing dynamic reconfiguration in component-based systems. In Ronald Morrison and Flávio Oquendo, editors, *EWSA*, volume 3527 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2005.

[DL06] Pierre-Charles David and Thomas Ledoux. Safe dynamic reconfigurations of fractal architectures with fscript. In *Proceedings of the 5th Fractal Workshop at ECOOP 2006*, Nantes, France, July 2006.

[Fra] Fractal ADL. *http://fractal.objectweb.org/fractaladl.*

[KC03] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[KM90] J. Kramer and J. Magee. The evolving philosophers problem: Dynamic change management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, 1990.

[MBC04] Rui S. Moreira, Gordon S. Blair, and Eurico Carrapatoso. Supporting adaptable distributed systems with formaware. In *ICDCSW '04: Proceedings of the 24th International Conference on Distributed Computing Systems Workshops*, pages 320–325, Washington, DC, USA, 2004. IEEE Computer Society.

[MK96] Jeff Magee and Jeff Kramer. Dynamic structure in software architectures. In *SIGSOFT '96: Proceedings of the 4th ACM SIGSOFT symposium on Foundations of software engineering*, pages 3–14, New York, NY, USA, 1996. ACM Press.

[OCL05] OCL 2.0 Specification. *http://www.omg.org/docs/ptc/05-06-06.pdf*, 2005.

[OMT98] Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor. Architecture-based runtime software evolution. In *ICSE '98*, pages 177–186, Washington, DC, USA, 1998. IEEE Computer Society.

[PV02] Frantisek Plasil and Stanislav Visnovsky. Behavior protocols for software components. *IEEE Trans. Softw. Eng.*, 28(11):1056–1076, 2002.

[TGGL82] Irving L. Traiger, Jim Gray, Cesare A. Galtieri, and Bruce G. Lindsay. Transactions and consistency in distributed database systems. *ACM Trans. Database Syst.*, 7(3):323–342, 1982.

E Composite Probes

JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering **©JOT, 2005**

Vol. 4, No. 3, May-June 2005

Composite Probes: a Generic Monitoring Framework for Hierarchical Management of Heterogeneous Data

Ada Diaconescu and Bruno Dillenseger, Orange Labs

Abstract

System monitoring has become an essential utility for managing software applications. However, as software systems are becoming increasingly complex, analyzing collected monitoring data is becoming progressively more difficult and costly a task. This article presents Composite Probes, a generic monitoring framework for complex system management. Composite Probes provides support for organizing heterogeneous data into hierarchical constructs that process data at different granularity and abstraction levels. Composite Probes represent building blocks that are instantiated, customized and connected to form flexible hierarchies adapted to various application requirements. System administrators configure each instance with specific data-processing functions, including aggregation, filtering and scheduling. A Composite Probes prototype was implemented and successfully tested on different distributed applications.

1 INTRODUCTION

System monitoring has become an essential utility for assisting the development, configuration and runtime administration of software systems. Generic and specific monitoring tools are being employed to test and calibrate software systems offline, as well as to supervise, manage and adapt software systems during runtime. Performance profiling, SLA-compliance verification and autonomic computing are only some of the important industrial and research areas heavily relying on monitoring functions. Existing monitoring utilities collect different data types from various managed entities (e.g. CLIF[1], LeWYS[2], Compas[3], Ganglia[4], or JVMTI[5]). Collected data can range from a system's hardware and software resource consumption, to a system's usage patterns and achieved quality attributes. Monitored data is subsequently analyzed in order to determine various

[1] CLIF: load-injection and monitoring framework for distributes applications (clif.objectweb.org)
[2] LeWYS: monitoring framework for hardware and software resources (lewys.objectweb.org)
[3] COMPAS: framework for performance management in J2EE applications (compas.sourceforge.net)
[4] Ganglia: distributed monitoring for high-performance computing systems (ganglia.sourceforge.net)
[5] JVMTI: JVM™ Tool Interface from Sun Microsystems (java.sun.com/j2se/1.5.0/docs/guide/jvmti)

properties, such as system correctness, performance, overall utilization, state, or SLA-conformance levels. However, as systems are becoming increasingly complex and monitoring tools more sophisticated, progressively larger amounts of heterogeneous data are being collected for analysis and diagnosis operations. Examining accumulated monitoring data is consequently becoming an increasingly complex and costly task, especially when periodically required during system execution. Available monitoring utilities generally provide data in the initially collected format and offer little support for processing this data. Consequently, massive amounts of heterogeneous data are represented at the same abstraction level, in a flat data structure. The responsibility for organizing, aggregating and filtering monitoring data is left entirely to data consumers, or clients. Meanwhile, multiple data-processing tasks are common to most clients, regardless of client-specific management goals. For example, various high-level indicators must be computed from low-level measurements in most complex management applications. There is an emerging need for aggregated data to be readily available, to represent high-level resource measurements or abstract indicators.

This paper proposes a novel monitoring framework called *Composite Probes (CPs)* that aims at extending current monitoring utilities with support for hierarchical data-organization and processing. CPs provides support for constructing flexible monitoring hierarchies from reusable and configurable probes. Such hierarchies can be configured to follow customized data processing and scheduling policies in order to aggregate and filter incoming data, possibly from heterogeneous resources, at multiple abstraction levels. In this manner, CPs instances in a hierarchy provide monitoring data at various granularity and complexity levels, representing basic or abstract resources, fine-grained measures or high-level indices. Namely, a CP can provide measures as different as a system's CPU usage, a cluster's overall resource load, a system's SLA-compliance level, or a an application's state. The main contributions of the CPs framework include:

1. Reusable applicative support for data assembly and event forwarding functions, commonly required for building data association and processing chains. Therefore, CPs helps decrease management costs by preventing replicated efforts, expertise and development work from being conducted at multiple sites.
2. Highly-customizable and extensible data-processing elements, that can be configured to use different aggregation, filtering and scheduling algorithms

Additionally, CPs features important characteristics required for a scaleable, manageable and adaptable monitoring framework:

3. Support for integrating and using low-level monitoring data from heterogeneous resources, at different system levels (e.g., OS, JVM, middleware and application).
4. Uniform data representation and control for all probe types
5. Standard external access via common communication and management protocols
6. Seamless extensibility, via new data-processing algorithms, new scheduling policies, new probe types and additional communication protocols
7. Support for integrating legacy probes and third party data processing functions
8. Inherent scalability of monitoring hierarchies, as large amounts of data can be collected and processed at multiple distributed sites.

2 COMPOSITE PROBES

Framework Overview

Composite Probes (CPs) is a generic monitoring and analysis framework for large-scale, distributed (LSD) applications. The framework's main goal is to provide support for organizing and managing massive amounts of monitoring data collected from heterogeneous resources. CPs aims at extending flat monitoring architectures with flexible hierarchical constructs, in order to facilitate data understanding, provide different information views and reuse data-processing functions.

CPs represent building blocks for constructing flexible and configurable hierarchies, which can process monitoring data at various granularity and abstraction levels. As depicted in Figure 1, in a CPs monitoring hierarchy data obtained from low-level system probes flows upwards through the hierarchy and undergoes incremental processing at each CP instance involved. CPs are classified into two major types, based on their roles and functions (Figure 1 and Figure 2). The first probe type is the *Basic Probe (BP)*, whose role is to extract monitoring data from the managed system resources. BPs represent *leaf* nodes in the hierarchy, meaning that they cannot be further composed of other probes. BPs process collected monitoring data and subsequently forward it to connected *parent* probes. The second probe type is the *Composite Probe (CP)*, whose role is to manage and organize incoming data from multiple data sources, or *child* probes. CPs can contain other CPs or BPs, which constitute the CP's *data sources* (Figure 4). CPs process incoming data and subsequently forward it to connected *parent* probes. Data processing in BPs and CPs involves data aggregation and filtering procedures, as dictated by well-specified scheduling policies. From an external perspective, clients have a uniform view of all probes. Probe access is restricted to probe external interface(s), which are identical for all probe types (Figure 2). For the scope of this paper, the term Composite Probes (CPs) will be used to indicate any of the two probe types, Basic or Composite, unless otherwise specified. The term also refers to the monitoring framework proposed. The exact meaning of the term will be clear from the used context.

All CPs can be identified via a unique *probe Id*. Clients use probe Ids to access any probe in a hierarchy, in order to retrieve monitoring data or send control commands. Communication can be done directly with a targeted probe, or indirectly, via the probe's parents. Indirect communication requires the path to the targeted probe to be provided as a request parameter, which allows the request to be routed to its destination probe.

Two main information flows characterize CPs hierarchies. These are the *data* flow and the *control* flow (Figure 1). In short, the data flow transports monitoring data upwards from lower-level BPs to higher-level CPs. The control flow transports control commands downwards from CPs at higher hierarchical levels to lower-level BPs.

The proposed framework has a modular, configurable and extensible design, which allows its main functions - aggregation, filtering, scheduling and communication - to be individually tuned and modified for each CP instance.

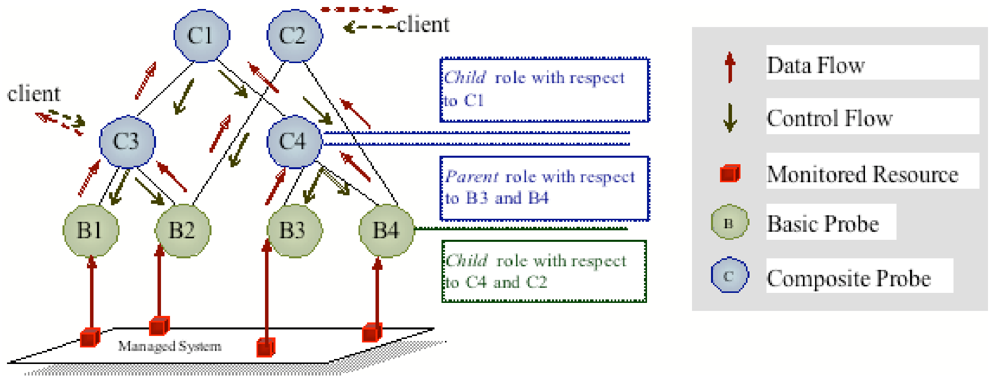

Figure 1: General view of a Composite Probes hierarchy

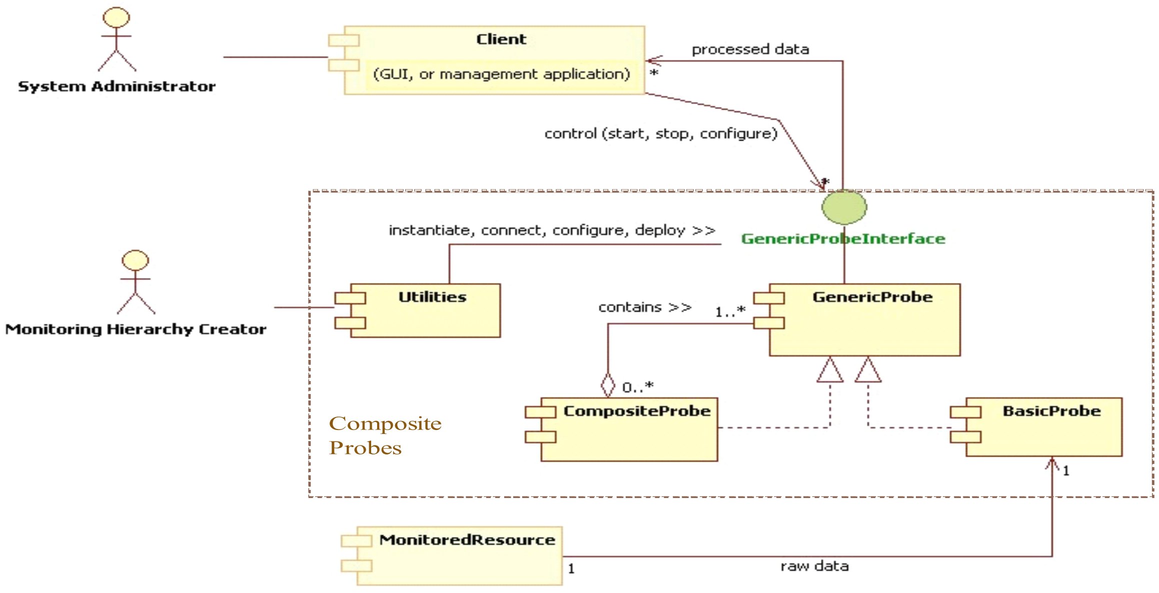

Figure 2: Logical view of the main component types provided in the Composite Probes framework

Example of Composite Probes Hierarchy for Cluster Monitoring

An example scenario in which the CPs use is potentially beneficial involves monitoring a distributed system, such as a computer cluster. Figure 3 provides a simplified example of such clustered system with two interconnected machines. For monitoring this system, two BPs are deployed on each machine for measuring their respective *memory* and *CPU* consumption. In a realistic scenario, low-level monitoring probes from tens or hundreds of clustered machines would produce a significant amount of fine-grained data, difficult to analyze manually during runtime. To alleviate such difficulties, this example shows how a CPs hierarchy is employed to aggregate monitoring data at different abstraction levels, such as overall system and cluster levels. The example CPs hierarchy in Figure 3 organizes monitoring data as follows. Two *system* CPs represent the overall load on each system (i.e. 'system1' and 'system2' probes). At a higher abstraction level, a *cluster* CP aggregates data from all available system CPs and provides cluster-level measurements (i.e. 'cluster' probe). Finally, a *cluster CPU* CP aggregates CPU data from all managed machines and represent the overall CPU load of the entire cluster. Hence, the cluster CPU probe simulates the existence of a single cluster CPU resource, hiding the fact that this abstract resource is aggregated from multiple system-level CPU resources.

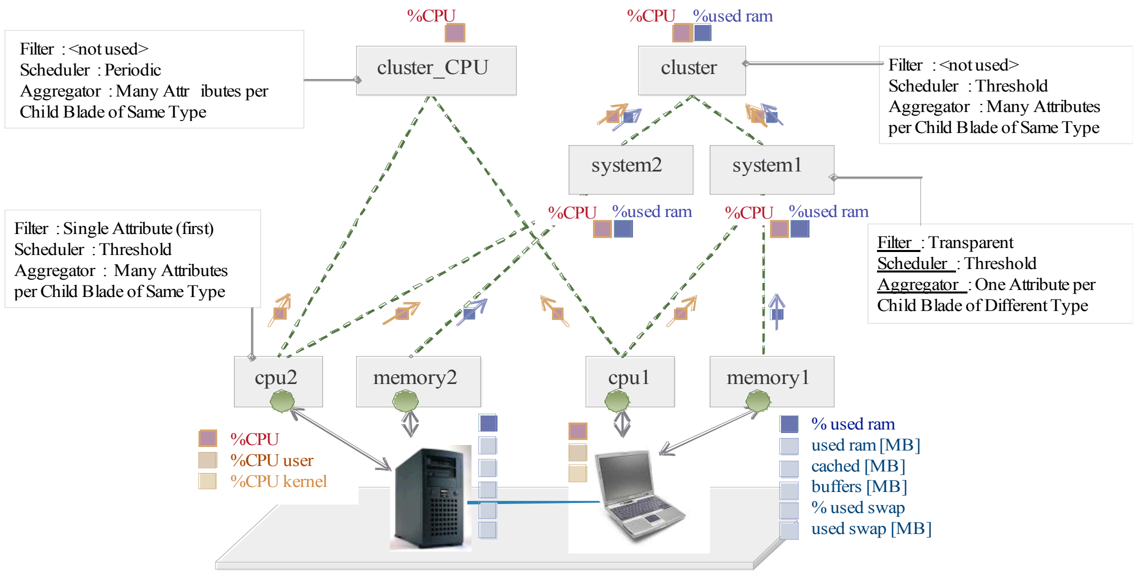

Figure 3: Sample Composite Probes hierarchy for monitoring a clustered system

Monitoring Data Flow

The monitoring *data flow* transports monitoring information through the CPs hierarchy. Data is collected by BPs and propagated up towards CPs at the top of the hierarchy. Figure 4 depicts a simplified view of the data flow through a monitoring probe. In short, a probe receives data from one or multiple *data sources*. Incoming data is collected, processed and stored as *local data*. From an external perspective, local data represents the data that a probe 'monitors'. For BPs, local data is based on actual measures taken from managed resources. For CPs, local data simulates measures taken from an abstract resource that the probe represents. In all cases, local data is subsequently filtered and forwarded to the probe's parents, or *data sinks*. In addition, external clients can access a probe's local data via direct method calls or by listening to the probe’s events (Figure 4).

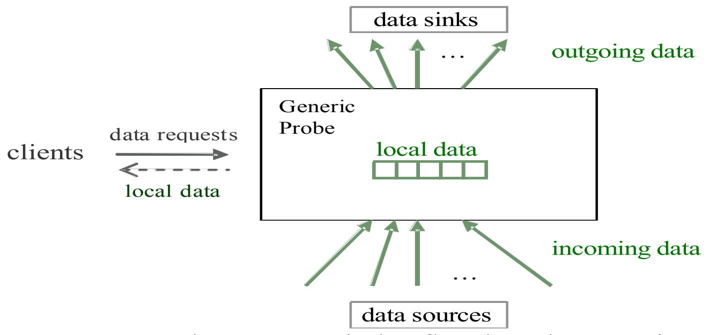

Figure 4: Generic data flow through Composite Probes

Figure 5 details the data flow view for BPs and CPs. The main dissimilarity between the two probe types is in the monitoring data *source*. Namely, BPs receive data from a single data source, which is an instrumented managed resource. The resource instrumentation code is represented in Figure 5 by the *Insert* component; it can be provided by the BPs or reused as existing legacy code. CPs receive data from multiple sources and aggregate this data into summary statistics or meaningful high-level measurements. Statistics are used to summarize a set of observations, in order to communicate concentrated or simplified information to external clients and parent CPs. Possible statistical functions include mean or median functions, standard deviation and variance, minimum or maximum functions. In addition, various aggregation algorithms can be specified for correlating data of different types into meaningful high-level indicators. For example, a system's congestion can be determined based on the individual hardware and software resource loads. Administrators are responsible for selecting or specifying the aggregation functions for each CP in the hierarchy. Aggregated results are stored as probe local data and can optionally be persisted to a storage support.

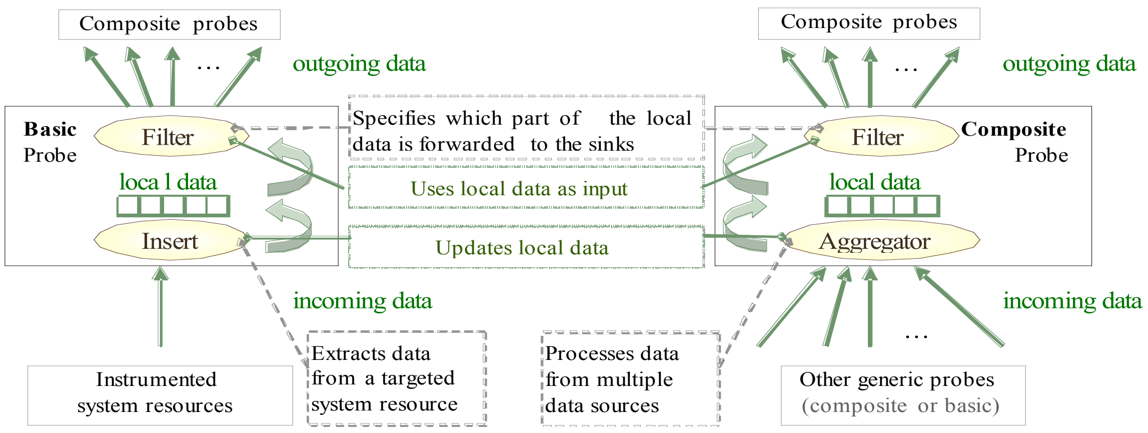
Composite probes
outgoing data
Composite probes
outgoing data
Basic Probe
Filter
loca l data
Insert
Specifies which part of the local data is forwarded to the sinks
Uses local data as input
Updates local data
Filter
Composite Probe
local data
Aggregator
incoming data
incoming data
Instrumented system resources
Extracts data from a targeted system resource
Processes data from multiple data sources
Other generic probes (composite or basic)

Figure 5: Data flow through Basic and Composite Probes

Control Commands

The *control flow* transports control commands for managing the probes' lifecycles. Namely, control commands are used to initialize, start, stop, pause, resume or terminate the execution of one probe or of a probes sub-graph. Namely, commands targeted at a certain probe can be optionally propagated to affect uniformly the probe's sub-graph. This facility simplifies hierarchy control processes by allowing an entire probe tree to be controlled via a single command sent to the tree's root.

Architectural Overview

BPs and CPs provide identical external interfaces and feature very similar internal architectures. The main architectural difference between the two probe types results from their different roles in a monitoring hierarchy. Namely, BPs use a specific *Insert* subcomponent for *extracting* monitoring data from the managed system. Conversely, CPs are in charge of *managing* monitoring data received from multiple lower-level probes. Consequently, CPs do not use an Insert subcomponent themselves, and obtain their data via connections to their child probes instead. Besides the manner in which they obtain their monitoring data, BPs and CPs contain identical subcomponents. Evidently, the way some of these subcomponents inter-communicate is influenced by whether a probe contains an Insert subcomponent or is merely connected to lower-level probes.

An overall view of the BP architecture is presented in Figure 6. The CP architecture is identical, except for the missing Insert subcomponent and additional connections to child probe components. The most important external interfaces include the *Probe Management*, *Data Collector Administration* and *Probe Control* interfaces. System administrators configure probes via their *Probe Management* interfaces. Supported management operations include setting and configuring a probe's aggregator, filter or scheduler functions, as well as connecting or disconnecting a probe from parent or child probes. Administrators use the *Probe Control* interface to perform control operations such as init, start, pause, resume, or stop on a targeted probe or probe sub-graph. Clients have access to a probe's monitoring data via the probe's *Data Collector Administration* interface. Probe interdependencies on external functionalities are represented via client interfaces. The principal client interface is the *Data Collector Write Delegate*, for forwarding data events to a probe's parents. Other client interfaces include the *Probe Response Delegate* and *Supervisor Information*, for sending asynchronous notifications, current probe state or abnormal events to parents.

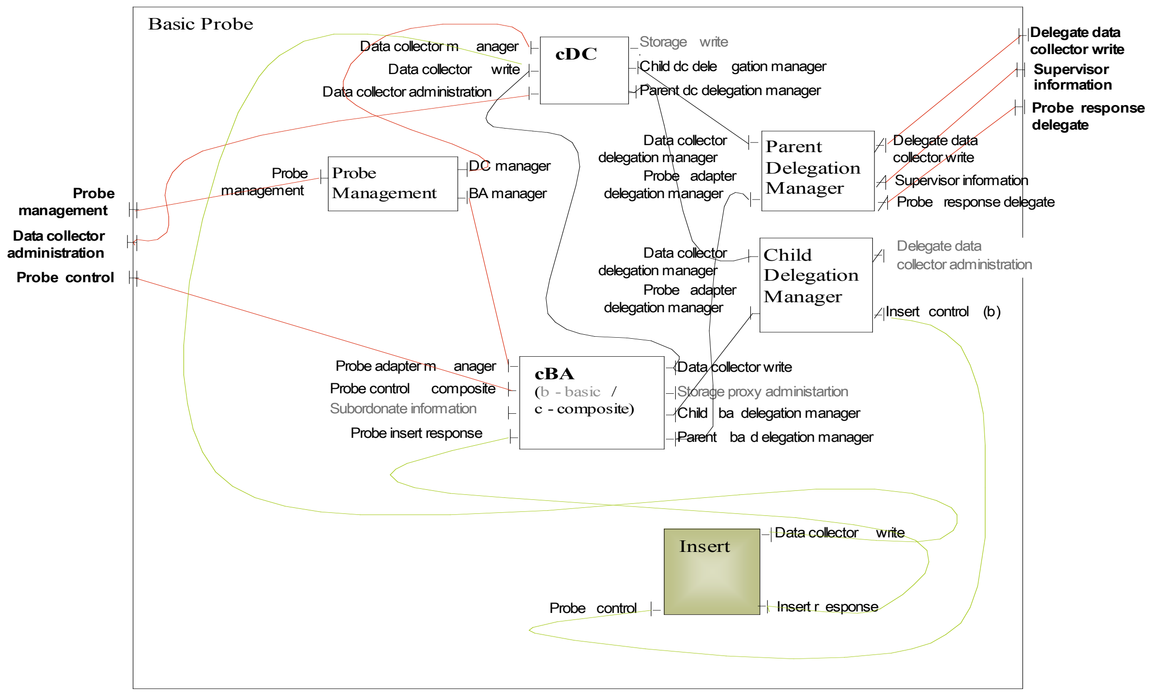
Basic Probe
cDC
Data collector manager
Data collector write
Data collector administration
Storage write
Child dc delegation manager
Parent dc delegation manager
Delegate data collector write
Supervisor information
Probe response delegate
Probe management
Data collector administration
Probe control
Probe Management
Probe management
DC manager
BA manager
Parent Delegation Manager
Data collector delegation manager
Probe adapter delegation manager
Delegate data collector write
Supervisor information
Probe response delegate
Child Delegation Manager
Data collector delegation manager
Probe adapter delegation manager
Delegate data collector administration
Insert control (b)
cBA
(b - basic / c - composite)
Probe adapter manager
Probe control composite
Subordonate information
Probe insert response
Data collector write
Storage proxy administration
Child ba delegation manager
Parent ba delegation manager
Insert
Data collector write
Insert response
Probe control

Figure 6: Basic Probe architecture (Composite Probe architecture is identical, except for the missing Insert)

Probe subcomponents common to both probe types are described as follows. The *Data Collector (DC)* subcomponent receives monitoring data and processes it according to specific aggregation, filtering and scheduling policies. Processed data is subsequently forwarded to the probe's parents and optionally sent to a storage support for persistent logging, or further processing purposes. The *Blade Adapter (BA)* subcomponent provides asynchronous support for executing the probe's control commands. This prevents clients from being blocked while their control commands are being executed, possibly on a significantly large probe sub-graph. The *Probe Management* subcomponent represents a portal for all management operations, redirecting requests to the specific subcomponents capable of processing them, and shielding external clients from internal details. Finally, the *Child* and *Parent Delegation Manager* subcomponents manage probe communication with child and parent probes, respectively. Their role is to isolate probe inter-connection logic from functional code, and facilitate eventual communication protocol changes.

3 DATA-PROCESSING IN COMPOSITE PROBES

A probe's *Data Collector (DC)* dictates the specific manner in which the probe handles incoming monitoring data and makes it available to clients and parent probes. The principal processes involved in collecting, processing and forwarding incoming data are assigned to specialized DC subcomponents, namely the *Aggregator*, *Filter* and *Scheduler*. Consequently, a DC's processing logic is highly configurable, as the algorithms and policies of each of its subcomponents can be individually specified and configured. In short, the DC Aggregator collects incoming data events over well-specified *intervals*. At the end of each interval, the Aggregator uses the collected data events to calculate and update the probe's local data. Intervals are dictated by the DC's Scheduler. Processed local data is sent to the DC's Filter and the filtered result forwarded to the probe's parent probes. This design enables administrators to create flexible hierarchies with custom data-processing paths by setting the data-processing policies of each instantiated probe.

Data Aggregator

The *Aggregator*'s role is to manage incoming data from multiple data sources. They provide two major functions, namely, collecting incoming data and processing it to calculate local data. The first function stores incoming data events according to a specified policy. For example, data received from identical probe types can be mixed together into a common storage (e.g. the 'cluster' and 'cluster CPU' probes in Figure 3), while data received from sources of different types must be stored separately (the 'system' probe in Figure 3). The second Aggregator function dictates how stored data is processed to calculate local statistics (e.g. maximum or weighted average functions).

Data Filter

The *Filter*'s role is to determine a probe's outgoing data for the probe’s data sinks. A typical filter determines which subset of a probe's local data will be sent as output data. A probe's Filter is unaware of whether the output data will be forwarded to one or multiple data sinks, as all inter-probe communication is managed by probe Delegation Managers.

Interval Scheduler

The *Scheduler*'s role is to dictate a CP's data processing *intervals*. Intervals dictate the exact instants at which a probe's Aggregator calculates local statistics based on the collected monitoring data. Each Scheduler has an associated *Task*, which can be implemented to trigger various data processing and forwarding functions. The Scheduler will trigger the Task's execution at the end of each interval.

4 CURRENT IMPLEMENTATION

Implementation Overview

A CPs prototype was implemented for demonstrating the main capabilities and functions proposed in the framework specification. The prototype implementation is based on Julia[6], a Java implementation of the Fractal[7] component model, and on additional Fractal utilities, including Fractal-ADL[8], Fractal-RMI[9] and Fractal-JMX[10]. Fractal is a hierarchical component model suitable for building complex applications with high modularity and adaptability requirements. The CPs framework is based on an existing monitoring and load-injection application called CLIF[11]. CPs extends CLIF's monitoring architecture and functions in order to add hierarchical data-management functions to the flat data representation initially supported. CPs equally reuses some of the instrumentation code provided in CLIF for monitoring UNIX, Windows and MacOS resources, including CPU, memory and disk utilization. The CPs prototype implements the BP and CP types, and provides several Aggregator, Filter and Scheduler implementations. The framework implementation supports remote client access to CP instances via the RMI[12] and JMX[13] protocols.

Implemented Aggregation Functions

Several Aggregator types are provided in the current CPs implementation. First, a "*Many per Identical Type*" Aggregator was implemented to aggregate multiple parameters from identical data sources. This implies that all data sources send data with identical formats and semantics. Secondly, a '*One per Different Type*' Aggregator was implemented to aggregate a single data item from different source types. This implies that each source sends a single parameter's values, where each parameter is of a different type. A third Aggregator type was implemented to process data on a managed element's state. The '*Component State*' Aggregator receives state events from a single element and calculates

[6] Julia: the reference Java implementation of the Fractal component model (fractal.objectweb.org/julia)
[7] Fractal: component model (fractal.objectweb.org)
[8] FractalADL: Fractal model's Architecture Description Language (fractal.objectweb.org/fractaladl)
[9] FractalRMI: enables remote method calls between Fractal components (fractal.objectweb.org/fractalrmi)
[10] FractalJMX: enables JMX management of Fractal applications (fractal.objectweb.org/fractaljmx)
[11] CLIF: generic Java-based performance testing framework (clif.objectweb.org)
[12] RMI: Java Remote Method Invocation from Sun Microsystems (java.sun.com/javase/technologies/core/basic/rmi)
[13] JMX: Java Management Extensions from Sun (java.sun.com/javase/technologies/core/mntr-mgmt/javamanagement)

its local data as the value of the last event in the interval. Finally, a '*Path State*' Aggregator was implemented to provide the overall state of an application call-path. Each Path State Aggregator collects state data from the multiple elements (e.g. nodes and connections) constituting the managed call path. At the end of each interval, the overall path state is calculated based on the individual element states.

Implemented Filtering Functions

Two filter types are available in the current CPs implementation. First, a '*Single Parameter*' Filter selects a single data element from the input data set and sends it as the output data set. Second, a '*Transparent*' Filter sends the entire, unchanged input data set as the output data set. This filter was provided in order to maintain a uniform data-processing architecture across all CPs.

Implemented Scheduling Policies

Two Scheduler types are available in the current CPs implementation. First, a *'Threshold*' Scheduler determines intervals based on the number of monitoring data events received. Namely, the scheduler dictates the end of an interval each time the number of received events crosses a certain threshold. The second scheduler type, a '*Timer*' Scheduler, determines intervals in a strictly time-based manner. Specifically, it uses a configurable period to determine the end of each interval. Both Scheduler implementations trigger the statistics calculation process in the associated probe Aggregator, at the end of each interval. New statistics are immediately filtered and forwarded to the probe's parents.

5 USAGE SCENARIOS AND PRELIMINARY RESULTS

Two example usage scenarios were tested to show the CPs framework benefits for monitoring large-scale, distributed systems. In the first example, a CPs hierarchy was built for monitoring various resource types in a clustered computer system. In the second example, a different CPs hierarchy was constructed to monitor the state of component-based data-processing applications, at various granularity levels. The goal of the two usage scenarios was to demonstrate the correct functionality of various CPs hierarchies and show the capacity of instantiating and inter-connecting different BPs and CPs, configured with various Aggregator, Filter and Scheduler types. The tested scenarios show the utility of using CPs when managing complex systems. Meanwhile, these initial scenarios did not explicitly test performance and scalability issues, even though such concerns were carefully considered in the framework's architecture specification.

Monitoring the Overall Load of a Clustered System

The CPs framework was used to create a monitoring hierarchy for supervising the resource usage of large-scale distributed systems. The monitoring hierarchy used is presented in the example in Figure 3. The following probe types were created to build the monitoring hierarchy in the presented scenario. First, two BPs were used to monitor the CPU and memory resources on the managed stations. One memory and one CPU probe was instantiated to monitor the respective resources on each machine. The BPs were configured to use Many per Identical Type Aggregators and Single Parameter Filters. Consequently, even though multiple CPU and memory parameters are monitored and made available at the BP level, only the values of one CPU and one memory parameter are forwarded to parent CPs. The percentage memory usage was selected as the unique forwarded measure for memory probes and the CPU percentage usage for CPU probes.

Secondly, a *system load* CP was prepared to represent the overall resource consumption on each machine. Two such system CPs were instantiated and used in the monitoring hierarchy, one for each managed station. Each system load probe collected data from the corresponding CPU and memory BPs on the monitored station. The system CPs were configured to use a One per Different Type Aggregator and a Transparent Filter. This implies that system probes collect one measure from each distinct child probe and forward all calculated values to parent probes. Concretely, a system probe collects two different measures, one from its CPU child probe and one from its child memory probe. Statistics calculated based on these measures are subsequently forwarded to parent probes, in this case, to a *cluster load* CP. In the current scenario, the system CP maintains the different collected measures separate from each other, as shown in Figure 7 (i.e. the 'system' probe in the bottom-right pane) and Figure 8 (i.e. 'system2' probe in the top pane). Nonetheless, the Aggregator used can easily be modified to calculate a unique *system load* measure, based on the separate measures collected. It is up to each system administrator to define the actual function to use for calculating a global system load measure based on discrete resource data (e.g., a max function considers the system load as equal to the load on the most used resource, most likely to become a bottleneck).

Finally, a *cluster load* CP was created to represent the overall resource load on the entire distributed system. A single probe of this type was instantiated and bound to collect data from all system CPs of the managed machines in the cluster. The cluster load CP was configured to use a Many per Identical Type Aggregator. This CP receives measures on the average memory and CPU consumption of the two systems in the cluster. Based on these measures, the Aggregator calculates the average CPU and memory consumption at the overall cluster level, at the end of each monitoring interval. The two selected stations had similar CPU and memory capacities, so that calculating the average resource usage in percentages for the two stations made sense. The Aggregator used can easily be modified to provide a unique *cluster load* measure, based on the separate CPU and memory data collected from the individual systems involved. In addition, a different Aggregator could equally be employed show the total resource usage in the cluster. Care must be taken when creating monitoring hierarchies from probes with different Aggregator and Filter types. System administrators should make sure that calculated measures at higher hierarchical levels make sense with respect to data received from probes in the lower hierarchical levels. All probes were configured to use Threshold Schedulers, meaning that statistics were calculated every time the number of collected monitoring events crossed a certain specified threshold.

Figure 7 and Figure 8 present snapshots of monitoring data obtained via the CPs hierarchy, at various abstraction levels. Figure 7 depicts data from monitoring probes deployed on the first machine and Figure 8 data from the second machine in the cluster.

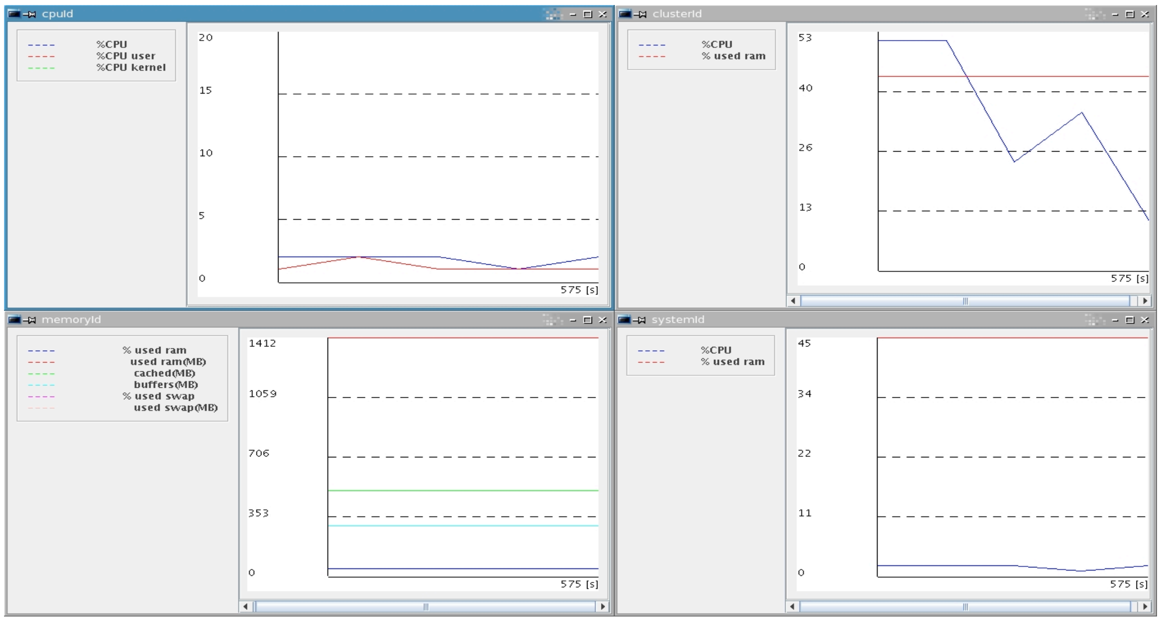

Figure 7: Composite Probes for System 1 and overall Cluster
Top-left: CPU 1 measures; bottom-left: memory 1 measures; top-right: cluster measures showing the average CPU and memory consumption in the entire cluster; Bottom-right: centralized system 1 measures

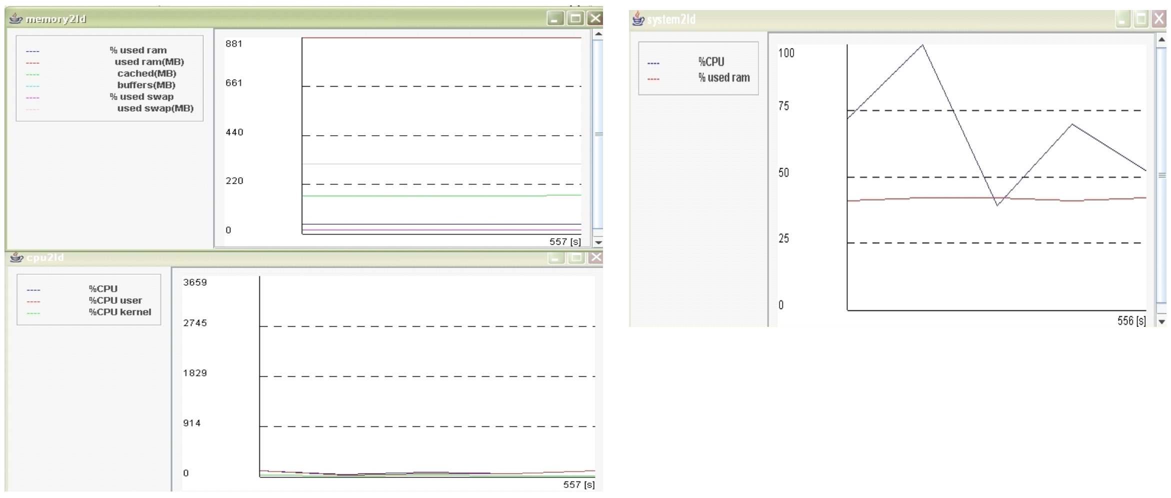

Figure 8: Composite Probes for system 2
Top: overall system 2 measures; middle: memory 2 measures; bottom: CPU 2 measures

As shown in the figures, the CPU, memory and system load probes were deployed on the respective stations they monitored. The cluster load CP was deployed on the first station. Monitoring data from the seven probes in the monitoring hierarchy is presented in the corresponding graphs in Figure 7 and Figure 8. Each graph is labeled with the unique id of the corresponding probe. The graphs show how the system probes provide centralized data on the overall CPU and memory consumption on the machines they represent. In addition, the cluster probe provides average consumption values for the managed cluster resources, based on values received from the constituent system probes. Initial tests were successfully carried out to verify the framework's support for external client access via the RMI and JMX protocols and for dynamic modifications on hierarchies of probes of similar types. Specifically, CPU probes were dynamically added and removed from a cluster CPU probe (Figure 3), without disrupting the monitoring system's execution.

Monitoring the Overall Availability of a Data-Processing Graph System

The CPs framework was used to create a monitoring hierarchy for supervising component-based data-processing applications. This type of applications involves several interconnected *nodes*, or components, that form a graph-like architecture. The application data-processing nodes are possibly distributed across multiple machines. Input data is forwarded between the graph's nodes, following a well-defined data-processing *path*. Each node contains a certain data-processing function, so that the node's output data is the result of the node's function applied on the node's input data. An application can provide several distinct processing paths, where different paths can share multiple nodes.

With respect to the performed testing scenarios, the central parameter of interest in the data-processing application scenario was the application *state*. State values were monitored at various granularity levels, such as node, data-processing path, overall node availability and global application availability. The possible node states are *Loaded*, *Initialized*, *Started*, *Stopped* and *Unloaded*. In order to display the nodes' states in graphical formats, the possible node states are mapped to numeric values (i.e. the Loaded state is mapped to value zero, Initialized to 1, Started to 2, Stopped to 3, and Unloaded to 4). A processing path's state depends on the respective states of the path's nodes. More complex scenarios can be envisaged to also consider inter-node connections and execution platform states when determining path states. Currently, a path is considered to be in the Stopped state if *any* of its nodes is in the state Stopped and in the state Started if *all* of its nodes are in the state Started. The *overall node availability* is calculated based on the individual states of all nodes in the application. Similarly, the *overall application availability* is considered based on the individual states of the available paths. Four probe types were used to build the CPs hierarchy for providing state information at the presented abstraction levels (Figure 9-a). With respect to the probes' internal configuration, each probe type was configured with a different Aggregator, depending on the probe's role and functions. Meanwhile, all probe types were configured to use the same Filter and Scheduler types, specifically, Transparent Filters and Timer Schedulers.

The different roles and functionalities assigned to each of the four probe types are described as follows. First, special-purpose BPs where employed to obtain information on the nodes' states. One BP was instantiated for each data-processing node in the monitored system (i.e., probes S1 to S10 in Figure 9-a). The BPs were configured to use State

Aggregators. In this testing scenario, the legacy instrumentation code provided by the data-processing application was used to provide the BP's Insert functionality. This was possible since the managed application already provided JMX-based monitoring events at the node level. Therefore, the BPs were implemented as JMX adapters to the existing instrumentation code, registering as listeners to node *state-change* events. At a higher abstraction level, a second CP type was used for representing the overall node availability in the application (i.e. the 'Ovrl Node States' probe in Figure 9-a). This CP is connected to all BPs (i.e., S1 to S10) and aggregates all individual node states for providing an overall view of the system node availability. A One per Different Type Aggregator was used for this CP type. Consequently, the 'Ovrl Node States' CP currently displays all monitored node states individually, providing a centralized view of the application state at node level. This Aggregator can easily be extended to use the centralized data and calculate the node availability, in percentages, as a unique measure of the system node state.

The remaining CP types are related to the application's data-processing paths. The third CP type monitors and represents the state of individual data-processing paths. A special Aggregator was implemented to calculate a path's state based on the individual states of the nodes in that path. This Aggregator can be extended to consider additionally the states of node inter-connections, or execution platforms. Finally, a fourth CP type was introduced to centralize all path states in the system (i.e. the 'Ovrl Path States' in Figure 9-a), which it collects from the individual path probes (P1, P2, P3 and P4). This CP type measures the global availability of the application, as experienced by external clients. The current implementation uses a One per Different Type Aggregator for this CP, separately maintaining the individual path states. This Aggregator could be extended to calculate the percentage of available paths, as a unique measure of the overall system availability.

The data-processing application depicted in Figure 9-b was instantiated and monitored using the CPs framework. This application consists of 10 data-processing nodes, interconnected to form four distinct data-processing paths. For example, path 1 in Figure 9 consists of the ordered nodes 1, 2, 3, 4 and 5; and path 4 consists of nodes 9, 3 and 10. As such, certain nodes are part of multiple data-processing paths. In the example, paths 1, 2 and 3 all share node 3, while node 10 is only involved in path 4. A direct consequence is that a node's failure can have different effects on the global system functioning, depending on the node's utilization in processing paths. For instance, a failure in node 10 will only disrupt the functioning of path 4, while a disruption in node 3 would affect all application paths and render the entire system unavailable. In this example, the dysfunction of a single node in the system, or 10% node unavailability, can have a considerably different impact on the *overall* system availability. More precisely, 10% unavailability at the node-level can result in 25% application unavailability if node 10 is disrupted and in 100% application unavailability if node 3 is disrupted. A monitoring hierarchy that highlights such information was constructed using the CPs framework, as shown in Figure 9-a. This CP hierarchy allows administrators to obtain system state information at various granularity and abstraction levels, as monitoring data is readily available on each node, path and global node and path availabilities.

Two example scenarios were considered, starting with a fully functioning application, where all nodes were in the 'Started' state. The first use-case scenario shows how stopping node 10 affects the local and overall application availability (Figure 10).

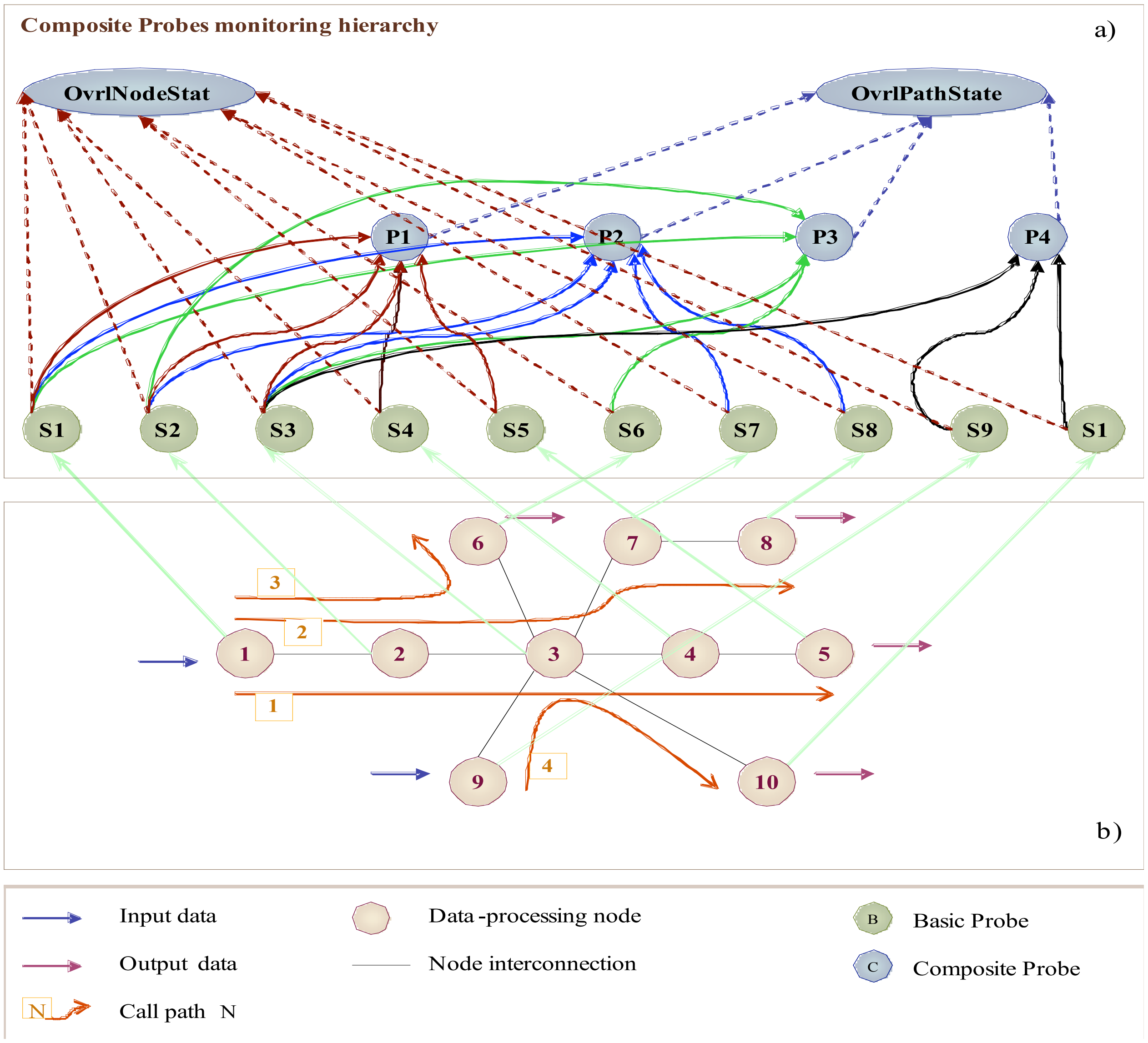

Figure 9: Using Composite Probes to monitor a data-processing application
a) CPs hierarchy, probe types: node BP (S1–S10), path CP (P1–P4), all paths CP (OvrlPathState), all nodes CP (OvrlNodeState); b) Monitored application: ten nodes (1-10) and four paths (1-4)

At the node granularity level, the monitoring data shows node 10 changing its state from Started to Stopped, while all other nodes remain in state Started. Path 4, which uses node 10, consequently changes its state from Started to Stopped, while all other paths remain unaffected. This data is depicted in Figure 10, as follows. The top-right graph corresponds to probe S10, for node 10; the top-left graph corresponds to probe ‘Ovrl Node States’ and shows centralized data on all nodes states; the bottom-right graph corresponds to probe P4 and shows the sate of path 4; the bottom left graph corresponds to probe ‘Ovrl Path States’ and shows centralized data on all paths states. The two overall Aggregators for probes ‘Ovrl Node States’ and ‘Ovrl Path States’ can seamlessly be extended to indicate the overall node availability of 90% (i.e. 9 out of 10 available nodes) and the global application availability of 75% (i.e. 3 out of 4 available paths). Similarly, the second scenario shows how stopping node 3 affects the local and overall application availability (Figure 11). In this case, the node's failure causes the entire application to become unavailable, as all paths are using node 3. At the overall level, this translates in a 90% overall node availability and a 0% global application availability.

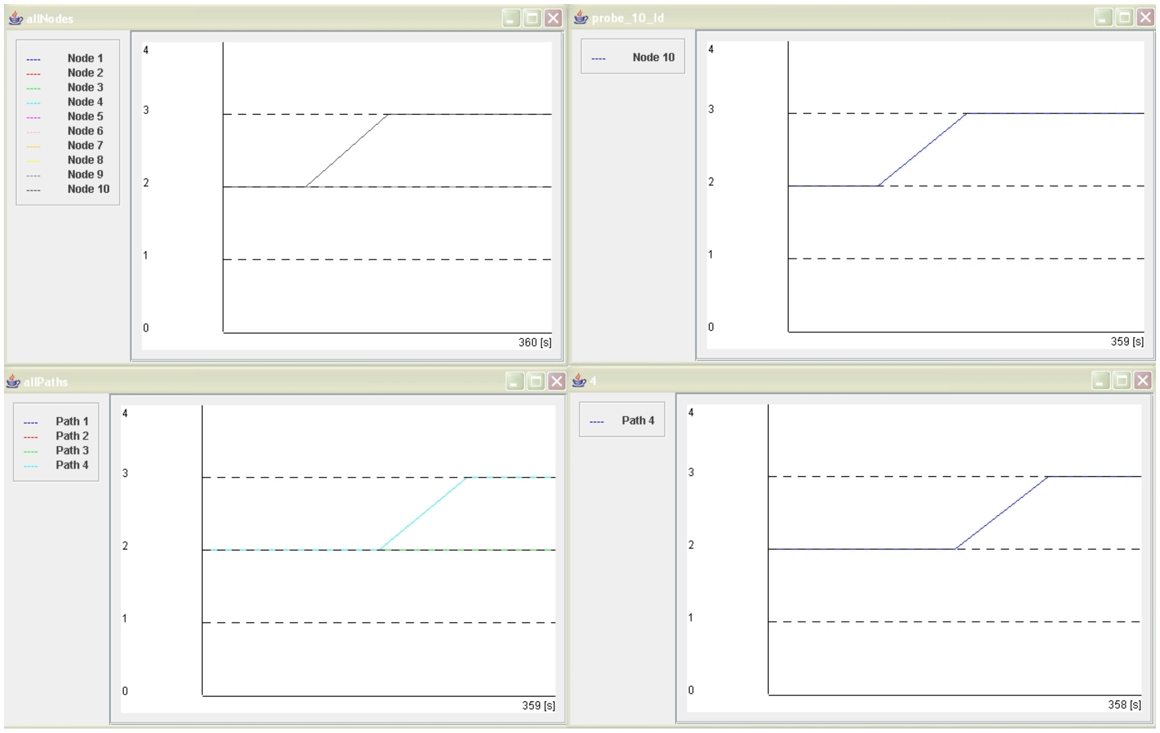

Figure 10: Local and overall impact on application availability when stopping node 10
Global node availability: 90%; Global data-processing application availability: 75%

The two scenarios indicate the important benefits of observing monitoring data at different granularity levels. While in both cases a single node was observed to fail, the actual impact on the overall application availability varied dramatically. Ultimately, from an application management perspective, the global data-processing availability is of major importance, as it is directly experienced by application clients. While significant, such global availability information is difficult to detect by merely following fine-grain measures at the individual node level. In addition, this difficulty increases dramatically with the application's scale and distribution. This example scenario shows how a CPs hierarchy can be used to alleviate such difficulties by providing aggregated monitoring information irrespectively of the system's scale. Similar CPs hierarchies can be equally employed to monitor the state and activity of different types of modular systems, such as component or service-based applications.

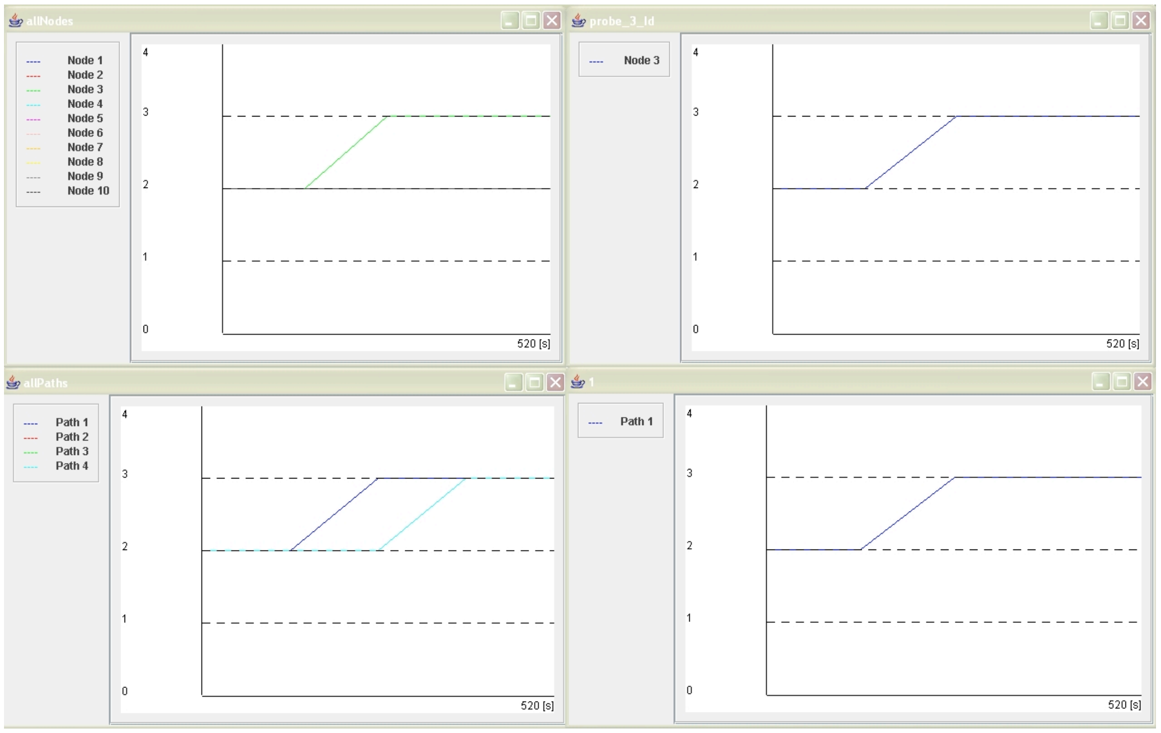

Figure 11: Local and overall impact on application availability when stopping node 3
Global node availability: 90%; Global data-processing application availability: 0%

6 RELATED WORK

Hierarchical monitoring systems, both freeware (e.g. Ganglia [1], Clumon[14], Supermon[15] [2], or Parmon [3]) or industrial (e.g. Big Brother[16], or Cluster Systems Management[17]) are available for monitoring clustered and grid systems. These tools represent mature, scaleable and efficient monitoring solutions for the precise system types they were designed for. On the Other hand, CPs' significant advantage lies in its high flexibility and extensibility features, as it provides support for creating and integrating customised probes and probe hierarchies for a wide range of system types.

A relatively recent project, Test & Performance Tools Platform[18] (TPTP) has many similar goals and characteristics with the proposed CPs framework. TPTP provides frameworks and services for developing test and performance tools, for system evaluation and profiling. The Monitoring Tools Project extends the TPTP platform to provide support for collecting, analyzing, aggregating, and visualizing data in detailed or

[14] Clumon: cluster monitoring system, NCSA (clumon.ncsa.uiuc.edu)
[15] Supermon: high performance cluster monitoring, Los Alamos National Laboratory (supermon.sourceforge.net)
[16] Big Brother : web-based system and network monitoring solution, Big Brother® Software (bb4.com)
[17] Cluster System Management: management of distributed and clustered IBM servers, IBM
[18] Eclipse Test & Performance Tools Platform Project (www.eclipse.org/tptp)

statistical views. Future work will follow the TPTP Project evolution and determine functionalities that can be reused and/or integrated with the CPs framework.

Virtually all research and industry projects in the autonomic computing area involve monitoring and analysis utilities for collecting, organizing and correlating data from the managed domain [4]. Depending on each project's goals, monitoring and analysis are required for creating system models (e.g. [5]), establishing execution contexts (e.g. [6]), evaluating and optimizing system performance (e.g. [6], or [8]), detecting application faults and system failures (e.g. [9], or [10]), or determining execution paths and performance anti-patterns (e.g. [11]). Most existing projects use proprietary solutions for collecting, grouping, aggregating and filtering monitoring data. CPs provides a reusable, scaleable and extensible framework for creating such monitoring and analysis facilities and accessing them via standard protocols.

7 CONCLUSIONS AND FUTURE WORK

Autonomic management systems require complex monitoring and analysis functions, which existing tools do not generally provide. This paper proposes Composite Probes (CPs), a flexible, hierarchical monitoring framework for autonomic management applications. CPs combines Basic Probes (BPs) that extract data from managed resources with highly customizable Composite Probes (CPs) that aggregate and filter data at various abstraction levels. CPs can be seamlessly extended with new instrumentation BPs, data-processing algorithms, scheduling policies and communication protocols. These characteristics make CPs suitable for a wide range of management applications and reusable across a wide range of system types. A CPs prototype was implemented and tested in two system management scenarios. The presented examples demonstrated how the CPs prototype could be used to create special-purpose monitoring hierarchies, combining the available aggregation, filtering and scheduling functions and integrating third-party instrumentation code via JMX. The examples showed CPs' suitability for monitoring dissimilar system types at various abstraction levels. Using CPs, managers have different views on a managed system, such as performance, architectural, or availability views, and browse through CPs hierarchies to determine the exact cause of an observed miss functioning. Future work will focus on integrating CPs with existing monitoring and management frameworks (i.e. CLIF, Jade and Jasmine). CPs will be extended as needed with new BP types, probe adaptors and data-processing functions. Additional support for JMS-based communication is equally envisaged.

8 ACKNOWLEDGEMENTS

The authors would like to thank their colleagues at Orange Labs, Marc Leger, Mehdi Kesis and Thomas Saillard for their valuable research ideas and technical support.

REFERENCES

[1] M. L. Massie, B. N. Chun, and D. E. Culler, "The Ganglia Distributed Monitoring System: Design, Implementation, and Experience", *Parallel Computing*, Vol. 30, Issue 7, July 2004

[2] M.J. Sottile, R.G. Minnich, "Supermon: a high-speed cluster monitoring system", *IEEE International Conference on Cluster Computing*, pp 39-46, 2002

[3] R. Buyya, "Parmon: a portable and scalable monitoring system for clusters", *Software Practice and Experience*, pp 723-739, 2000

[4] "An Architectural Blueprint for Autonomic Computing", *IBM White Paper*, 2005 www-128.ibm.com/developerworks/autonomic/library/ac-summary/ac-blue.html

[5] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste, "Rainbow: Architecture-Based Self Adaptation with Reusable Infrastructure,", *IEEE Computer* Vol. 37, Num. 10, pp. 46-54, October 2004

[6] A. Diaconescu, J. Murphy, "Automating the Performance Management of Component-Based Enterprise Systems through the use of Redundancy", *ACM/ IEEE Conference on Automated Software Engineering,* Long Beach, USA, , 2005

[7] A. Diaconescu, A. Mos and J. Murphy, "Automatic Performance Management in Component Based Software Systems", *International Conference on Autonomic Computing*, New York, USA, 2004

[8] S. Bouchenak, N. De Palma, D. Hagimont, S. Krakowiak, and C. Taton, "Autonomic Management of Internet Services: Experience with Self-Optimization", International Conference on Autonomic Computing, Dublin, Ireland, 2006.

[9] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, E. Brewer, "Pinpoint: Problem Determination in Large, Dynamic Internet Services", *International Conference on Dependable Systems and Networks*, pp 595 – 604, 2002

[10] S. Bouchenak, N. De Palma, D. Hagimont, and C. Taton, "Autonomic Management of Clustered Applications", *IEEE International Conference on Cluster Computing*, Barcelona, Spain, 2006

[11] T. Parsons, J. Murphy, "Detecting Performance Antipatterns in Systems Built using Contextual Component Frameworks", *Journal of Object Technology*, to appear.

[12] B. Dillenseger, E. Cecchet, "CLIF is a Load Injection Framework", *Middleware Benchmarking workshop*, *OOPSLA*, Anaheim, CA, USA, 2003

About the author(s)

Ada Diaconescu is a research engineer in the Adele/LSR group, University Joseph Fourier, Grenoble, France. The presented work was carried out as part of her postdoctoral work at Orange Labs. She obtained her PhD from the School of Electronic Engineering and Computing at Dublin City University. Her main research interests include autonomic computing and complex systems. Contact her at adadiaconescu@gmail.com. See also adadiaconescu.there-you-are.com

F ECA rules for Components

Flexible Reactive Capabilities in Component-Based Autonomic Systems

Jayaprakash Nagapraveen[*]
HADAS Group, LIG
Saint Martin d'Hères, France
nagapraveen.jayaprakash@imag.fr

Thierry Coupaye
France Telecom R&D
Grenoble, France
thierry.coupaye@orange-ftgroup.com

Christine Collet
INP Grenoble
LIG Laboratory
Saint Martin d'Hères, France
Christine.Collet@imag.fr

Pierre-Charles David[†]
OBASCO Group, EMN/INRIA
Nantes, France
pcdavid@gmail.com

ABSTRACT

Reactive behaviour, the ability to (r)eact automatically to take corrective actions in response to the occurrence of situations of interest (events) is a key feature in autonomic computing. In active database systems, this behaviour is incorporated by Event-Condition-Action (ECA or active) rules. Our approach consists in defining a mechanism for the integration of these rules in component-based systems to augment them with autonomic properties. The contribution of this article is twofold. First, we propose a rule model, i.e. a rule definition model and a rule execution model, that can be coherently integrated into a component model. Second, we propose a graceful architecture for the integration of active rules into component-based systems in which the rules as well as their semantics (execution model, behaviour) are represented/implemented as components, which permits i) to construct personalized rule-based systems and ii) to modify dynamically the rules and their semantics in the same manner as the underlying component-based system by means of configuration and reconfiguration. These foundations form the basis of a framework/toolkit which can be seen as a library of components to construct events, conditions, actions, rules and policies (and their execution subcomponents). The framework implementation is extensible: additional components can be added at will to the library to render more elaborate and more specific semantics according to certain applicative requirements.

[*]This work was done while the author was a PhD student at France Telecom R&D and IMAG-LSR

[†]This work has been done while the author was a post-doctoral fellow at France Telecom R&D.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architecture; D.2.13 [**Software Engineering**]: Reusable Software— ; H.2 [**Database Management**]: Systems

General Terms

Design, Experimentation

Keywords

autonomic systems, component-based architectures, ECA / active rules

1. INTRODUCTION

The overall motivation which underlies the emergence of autonomic computing[9] is based, from an IT industrial point of view, on the observation that the costs related to the software infrastructures (TCO) currently move from costs related to the development and licensing to costs related to the deployment and exploitation. It appears also quite clearly that a "manual administration" of pervasive environments such as "machine to machine" (automotive, home networking, etc.) software infrastructures or grid computing is, in practice, almost impossible. Autonomic computing thus aims basically at, as much as possible, automating the deployment and management (administration) of software systems in order to decrease human interventions and associated costs.

We consider in this work that an autonomic system is composed of an *autonomic infrastructure* superimposed on a *target component-based system*. The upper layer or the autonomic infrastructure, is responsible for implementing a *control loop*, i.e. instrumenting the components of the target system for monitoring, detecting and notifying events, diagnosing the system based on these events, and making decisions to determine what and how corrective actions need to be executed, and finally, executing the corrective actions on the components of the target system.

Our work focuses on the architecture and the behaviour of control loops. We propose to use Event-Condition-Action (ECA or active) rules[12], a mechanism widely used in active database systems to provide a reactive behaviour (an elaborated form of *triggers* as found in most commercial DBMS). The fundamental idea is to "extract" the reactive functionality of active database systems[8], and to "adapt and inject" it in component-based systems so as to provide them with autonomic capabilities.

Our objective is to realise a modular and extensible framework/toolkit for the construction of ECA rule-based autonomic architectures. In this framework, the rules as well as their semantics (execution model, behaviour) are represented/implemented as components, which permits i) to construct personalized rule-based systems and ii) to modify dynamically the rules and their semantics in the same manner as the underlying component-based system by means of configuration and reconfiguration. In its basic version, the framework is a library formed of basic components (and subcomponents) which permits to construct basic rules. The framework is extensible, i.e, additional components can be added at will to the library to render more elaborate and more specific semantics according to certain applicative requirements.

The main goal of this article is to introduce the design of this framework (called Fractal ECA) through an illustrative example used throughout the paper. Due to space constraints, only the elements of the framework required for the comprehension and implementation of the example are discussed.

Sample Autonomic Scenario. Let us consider as a target system, a minimal HTTP webserver with the sole functionality of retrieving HTML documents. It is made of two main components, namely a *Request Receiver* and a *Request Processor*. The request receiver component or front-end uses a lower-level scheduler component that creates a new activity (a thread in this case) for each request. The request processor component or back-end analyses the request, logs it and responds to it. Our Comanche web server[1] (shown in the bottom of Figure 4), is a multi-threaded system in Java that follows a prefork model - the parent process forks new child thread for every request. Most of the action takes place in the child threads - figuring out what the requests mean and sending the requested content back to the client. Once the processing is done, the child thread waits for the next one from its parent. Multithreading avoids the server from being idle until an I/O operation is finished. On the other hand, it introduces an overhead on the CPU due to the creation of new threads and to the commutation between existing threads. The right trade-off between these two conflicting factors reflects on the system performance. We propose a simple active rule to maintain this compromise (in complete informal format):

RULE ***DoubleThreadPoolSize***
ON each webpage request
IF (no free threads in the thread pool)
DO { Double the thread pool size }

[1]which comes with the Fractal distribution. See http://fractal.objectweb.org.

The above rule's purpose is to extend the capacity of the webserver by increasing the size of the thread pool, that is the number of possible concurrent child threads. A complementary rule can be equally envisaged that reduces the size of the thread pool, when the number of passive threads are high, thus liberating unused ressources.

The overall semantics of an ECA rule is the following: *when an event of type E occurs, if condition C is satisfied then execute action A*. Behind this apparent simplicity hides a great deal of complexity that raises many questions which we shall illustrate based upon our concrete example. Indeed, several questions are raised by the execution of this simple rule, to list a few:

- Is a new instance of a rule triggered for every occurrence of a triggering event (webpage request)? or once for several occurrences of the triggering event?
- When are the condition and action parts executed with respect to the target system execution? before the execution of the triggering operation or after?
- Is the rule executed in the same execution thread as the triggering operation or in a separate one?

For the above questions, several options exist, each representing an execution strategy of the rule. In the sequel, we discuss possible answers to these questions based on our illustrative scenario. More generally, our core work is to provide the rule programmer with an abstract framework for reasoning about rules execution and an actual architectural framework for practically programming rules and their semantics.

The rest of the paper is organized as follows. Section 2 and 3 introduce our first contribution: the proposition of an ECA rule model suited for component-based systems made of a rule definition model (Section 2) together with a rule execution model (Section 3). Section 4 and 5 introduce our second contribution: the proposal of an architectural design that allows for the graceful integration of active rules into component based-systems (Section 4) and some elements about the actual implementation of rules as Fractal components (Section 5). Section 6 discusses related works. Section 7 concludes the article.

2. RULE DEFINITION MODEL

A definition (or knowledge) model specifies how the rules are represented and manipulated. This section describes the rule definition model we propose for component-based autonomic systems.

Event. An event is a happening of interest at a given point in time. It is characterized by an event type, i.e, an expression describing a class of significant occurrences of interest. In our framework, we consider the following event types: i) *applicative* - corresponds to inter-component interactions, ii) *structural* - represents modifications (reconfigurations) of the structure (topology) of the system, like adding or removing a component, or creating new bindings between components, iii) *system-level* - characterises events coming from the external or underlying environment or context of execution (e.g. JVM and OS events). In our illustrative scenario, we have an applicative event type: an operation (method) invocation on a component interface, and more precisely calls on the interface of the frontend component of our web server to request web pages.

Condition. Conditions are optional and express additional constraints on the state of the system that must be satisfied for the action part to be executed. For example, in our autonomic scenario, the condition predicate checks the number of available threads, These kinds of condition expressions (e.g. whether an attribute's value is bigger than a particular value or not) are simple boolean expressions built using logical operators. More complex expressions can be formed based on queries on the structure of the system as well as on its behaviour. A query that selects the various components linked to a particular one is one such example. In such a case, the condition is considered to be true when the query returns a non-empty result.

Action. The corrective (re)actions that the target system can be subjected to are expressed in the action part of the rule. The event and condition parts of the rule serve to analyse the symptoms affecting the system. In our scenario, a method call to the Request Receiver component triggers the rule. Its condition part evaluates if the number of free threads is below a limit. If yes, the action that increases twofold the size of the thread pool is performed. To rectify the anomalies, the action can range from simple parameterizations of component attributes, for example, an increase in the size of a cache or pool, to complex structural reconfiguration operations, which can include addition, removal or replacement of one or several components. Other types of actions can be envisaged, like external notifications, for example, an email or SMS notification to an administrator.

3. RULE EXECUTION MODEL

A rule execution model specifies the behavioural semantics of rules. This section introduces the design of our proposed execution model and discusses the main dimensions of this model on our illustrative scenario.

The entire execution of a single rule is comprised of the following three phases and various states:

1. Triggering and Event Processing Phase R(E): this phase begins with the notification of the event(s) that triggers ("wakes up") the rule. The notification is performed by the entity on which the event occurs. It consists in processing the event(s) based on the various rule execution parameters. The rule goes from the *triggerable* state to the *triggered* state.
2. Condition Evaluation Phase R(C): the second phase of the execution evaluates the condition expression. If the condition is satisfied then the rule transits from the *evaluable* state to *evaluated* state.
3. Action Execution Phase R(A): the last phase of the rule execution corresponds to the execution of the action part of the rule. It takes the rule from the *executable* state to the final state of *executed* state (generally confonded with the initial *triggerable* state), thus inducing a positive feedback change in the system behaviour.

It is worth mentioning that the condition of a triggered rule is not always evaluated immediately (hence the two separate states *triggered* and *evaluable*), and that a triggered rule with a satisfied condition is not always executed immediately (hence the two separate states *evaluated* and *executable*). When and how (e.g. which activity/thread) a rule is processed depends on the various dimensions of the rule execution model. Some of the most important ones are discussed later in the context of our illustrative scenario. Of course, when multiple rules are concerned, which is the case in real autonomic systems, an execution model also specifies when and how rules triggered simultaneously (by same or different events) (a.k.a. *multiple rules*)and rules triggered by other rules (a.k.a. *cascading rules*) are executed. This is handled by *rule execution strategies* (or *policies*) which basically specify the scheduling of rules (e.g. depth-first order, width-first order, flat order, by cycles in sequential or parallel settings). Due to space limitation, this article does not detail these aspects. The reader may refer to [4]. Prior to that and more fundamentally, if rules have an execution model of their own, it has to be stated that the introduction of active rules in a system (be it a database or a component-based system) has also a non negligible impact on the behaviour of that system. Indeed, there exists a dependency between the execution of the system and the execution of rules, for it is the former that triggers the latter and also the two executions are interwoven/intertwined together.

3.1 From active database systems to active component based systems

Active rule execution models in database systems have been extensively studied but cannot be directly applied to component-based systems. First, events that trigger a rule in an active DBMS are query (SQL) statements on a global data schema, so are the condition and action parts. But this it is not the case in component-based systems where we have a variety of events, condition predicates and actions (as defined in the rule knowledge model). In active database systems, all rule operations are performed on a single database, whereas in a component-based system, they may have to be performed on different components. Indeed, in a component-based system, situations of interest can happen on any component of the system. To gain a thorough understanding of the component and its execution environment, we might have to perform additional queries on it or on its neighbours and finally execute the corrective actions elsewhere. So, the distributed characteristic of component-based systems is one of the distinguishing factors.

Execution units and execution points in component-based execution models. Finally, besides the two differences we have just mentioned, a key difference between active database systems and active component-based systems is that execution models in active database systems are based on a central concept, that of *transaction*, which is (generally) inexistant in component-based systems. A transaction in database systems is a sequence of operations that constitutes a unit of concurrency and recovery thanks to the well-known "ACID properties" (*Atomicity, Consistency, Isolation and Durability*). Transaction is a core and foundational concept of active database systems because, thanks to transaction *demarcations* (*start, commit, abort/rollback*), they provide a natural and convenient execution unit for the execution of active rules. An execution unit specifies an interval (between two execution points in a sequential flow or basically between two points in time) during which events can be detected/notified to interested rules and rules can be evaluated and executed. Hence, an execution unit specifies the *granularity* of rules execution. On the one hand, component-based systems generally do not consider transactions. On the other hand, the behaviour of a component-based system generally refers to interaction through interfaces only, thanks to operation (methods in Java) invocation. Hence, we define the execution unit in component-based systems as delimited by the interval between the reception of an operation invocation on a server interface and the emission of a response onto a client interface. For method invocations on a component's functional interfaces (produces applicative events), and operations that modify the structure of the system (produces structural events), we may signal two events: *begin* and *end*. Other forms of events (e.g. system events) can be integrated in the model that by considering their begin and end events are merged (i.e. they both represent the same execution point or point in time).

3.2 Rule execution dimensions

Based on these hypotheses, we consider our approach for defining a rule execution model, similar to the one followed in active database systems[4], where it is defined as a set of *dimensions*, with each dimension being attributed a particular *value*. The differences/issues outlined above, have been addressed in the form of a flexible rule execution model for component-based systems, adapted from rule execution models defined in active DBMS. The sequel discusses rule execution dimensions in the context of the autonomic scenario introduced earlier.

Event Processing Mode. On every webpage request, the Comanche webserver requests the scheduler service for an execution thread. So, if an instance of the thread management rule is triggered for every call, then the system ressources would be spent unnecessary resulting in a lower performance. Ideally, a rule needs to be triggered once at the appropriate moment to rectify the situation. The event processing dimension addresses this issue, with the possibility of triggering a rule for several occurrences of the event type. A rule may handle either only one event at a time or a set of events. This is specified in the *event processing mode*, having an *instance-oriented* semantics for the former, and a *set-oriented* semantics for the latter. In other words, an instance-oriented semantics suggests that a rule will be triggered for every occurrence of a triggering event. Such a kind of event processing strategy is interesting whenever each event has to be treated individually, e.g. when an exception is raised, or on every attack or on every forced entry by a malicious user which requires preventive measures to be triggered in the form of a rule. If several rules are triggered with the same purpose, system resources are bound to get depleted, affecting the performance of the application. Therefore, this strategy is beneficial when a single execution of a rule is enough to resolve the anomaly in the system. We shall opt for a set-oriented value to the rule's event processing mode: as said earlier, a single execution of the rule is sufficient to retain the performance level of the system.

Coupling Modes. Once a rule is triggered, we have to determine when and how it will continue its execution for it should not affect the system's execution. If we consider our thread management rule, should our rule be executed before processing the request or after? Should it be done in the same execution thread or in a separate one? Several options exist for the above. This is taken care of by the *coupling mode* dimension characterized by the couple : < *execution mode, activity mode*>.

The *execution mode* specifies when the condition and action parts of a rule are evaluated and executed with respect to the execution of the triggering operation. The triggering operation is the method invocation on a component's interface which produces events that trigger a rule. Such a rule that is activated and is ready for execution, is called a triggered rule. The commonly supported execution modes include (cf. Figure 1):

- *immediate (or as soon as possible - ASAP):* the triggered rule is processed immediately - its condition is evaluated, if true, the action is executed without any pause.
- *defered:* the triggered rule is processed later, awaiting the end of the triggering operation. The rule is triggered when an event indicating the beginning of the triggering operation is received. But the condition evaluation and action execution of the rule are processed only on receiving an event indicating the end of the triggering operation.
- *delayed:* here, the condition is evaluated immediately after the rule has been triggered, but the action part is executed only on the completion of the triggering operation.

Rules dealing with security issues have generally a high priority, and may typically take the *immediate* value in order to execute instantly before the damage is done. *Defered* rules are typically used in situations in which the action has to be executed on the final state of the system. In this respect, immediate rules might embed pre-conditions, while defered rules might embed post-conditions. *Delayed* rules are intermediary with a condition evaluated at the beginning of the triggering operation (i.e. before the state of the system might be changed by other rules for instance) and the action executed at its end. Our rule concerns the performance of the system. It permits the system to take preventive measures in order to maintain its performance levels. So, it is not so crucial, and can be executed once the initial task is completed, a defered value will be attributed to our rule's execution mode.

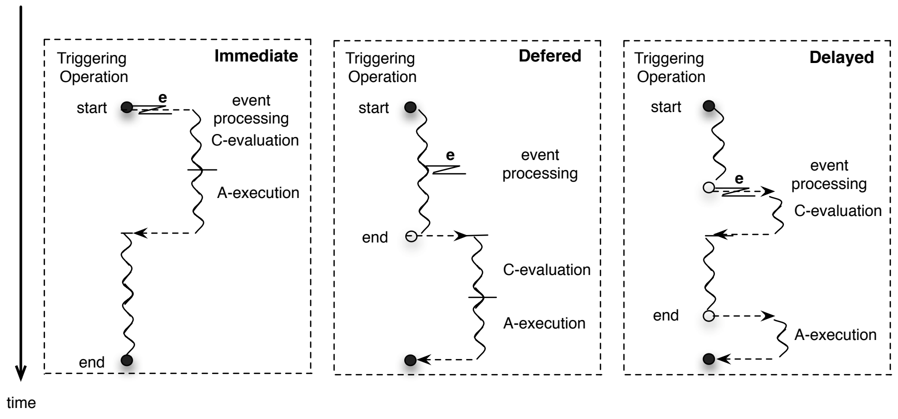

Figure 1: Execution Mode

An execution thread represents a sequential flow of control. The *activity mode* dimension determines whether the triggered rule is executed in the *same* thread as the triggering operation or in a *separate* thread. It is recommended to create a separate flow of control for aspects related to the administration of the system, e.g, logging. Thus, separating the system under control and its administration, so that its normal execution is not too disturbed. On the other hand, some administration scenarios might require the system to be paused, to enable some modification, and to resume later its execution. Depending upon the scenarios, the best method of executing the rule has to be judged and employed. For our thread management rule, we shall follow the conventional choice of executing the rule in a separate thread because there is no risk due to concurrency on the usage of threads in the pool since the rule creates only new threads.

Focus on interactions between a couple of dimensions. Since there exist some dependancies and intricacies among these dimensions, the individual semantics of the dimensions might slightly differ which one could guess at a first glance when considered as a whole. To illustrate such intricacies, we now focus on the interactions between event processing and execution modes.

Prior to presenting the possible combinations for the couple $<$ event processing mode, execution mode $>$ (including the ones that match the above choice in our example) , the following notations are employed in the sequel and in Figure 2:

- b_n: the event corresponding to the beginning of the n_{th} method invocation
- e_n: the event corresponding to the end of the n_{th} operation (method) invocation
- $R_n(E)$: depicts the event processing phase of the n_{th} occurrence of the considered triggered rule R
- $R_n(C)$: depicts the condition evaluation phase of the n_{th} occurrence of the considered triggered rule R
- $R_n(A)$: depicts the action execution phase of the n_{th} occurrence of the considered triggered rule

The events b_n and e_n are representative of the n_{th} occurrence of a triggering event type.

1. $<$ Immediate, Instance $>$: on every event b_n, a rule is triggered and processed immediately in its entirety. All e_n events are ignored.

2. $<$ Immediate, Set $>$: on the event indicating the beginning of the first triggering operation, i.e, on b_1, a rule is triggered and continues processing till its completion. All b_is that occur till the triggered rule completes evaluation, are consumed by the rule, i.e, the triggering operations are taken into consideration by the rule in execution. The next b_i when the rule is in $R_n(A)$ or after, triggers the next rule. Similarly, this rule also consumes all b_i that occurs till it completes evaluation. Two similar rules can coexist when one is in the action execution phase and the other begins execution.

3. $<$ Defered, Instance $>$: on every b_n, a rule is triggered, it processes the event and waits for a complementary e_n event to continue evaluating - $R_n(C)$ and complete execution - $R_n(A)$.

4. $<$ Defered, Set $>$: all b_is trigger each a rule. The rules complete the $R_n(E)$ phase, and wait for an end event e_i. When a e_i event is received, all rules triggered after R_i are discarded and their corresponding triggering operations consumed by the rule R_i. All events e_j where j $>$ i received later, are discarded. For any e_k where k $<$ i, the corresponding rule R_k resumes execution, and consumes all triggering operations that have triggered rules R_j where k $<$ j $<$ i.

5. $<$ Delayed, Instance $>$: on every b_n, a rule is triggered, processes its event part - $R_n(E)$, evaluates its condition part $R_n(C)$ and waits for the e_n event to execute the last part of the rule - $R_n(A)$.

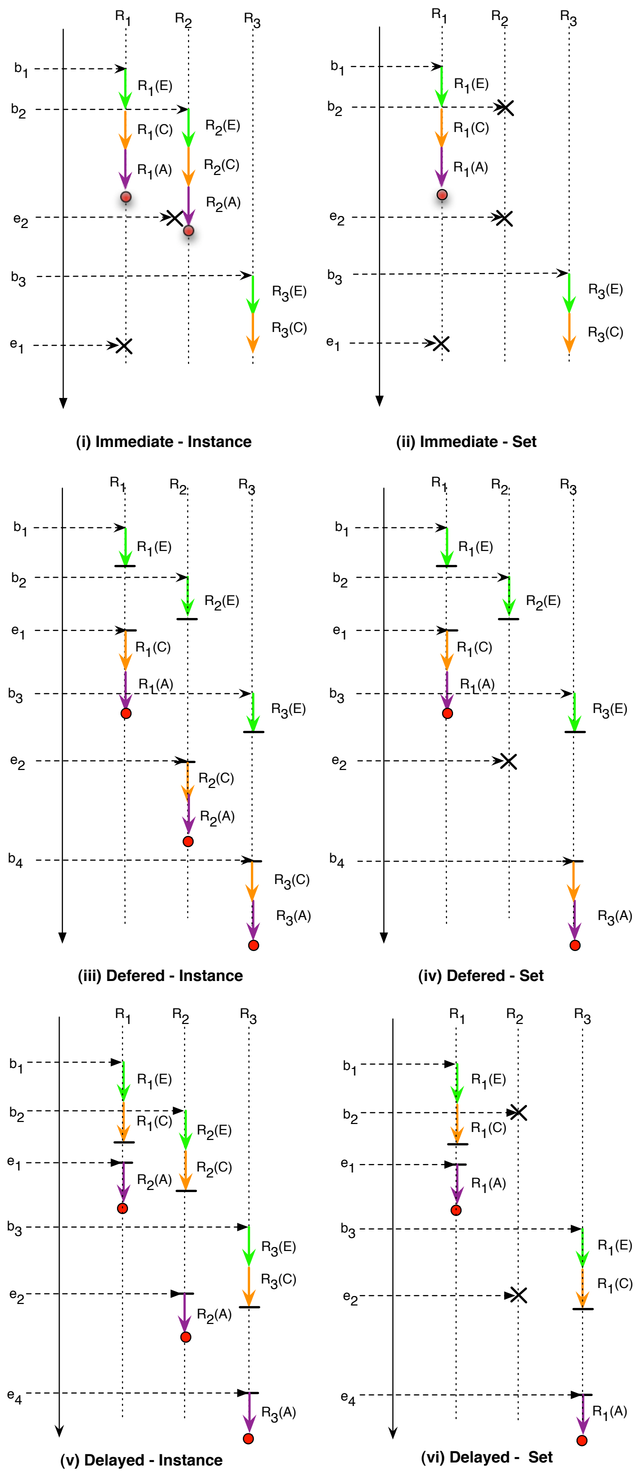

Figure 2: Execution Models

6. $<$ Delayed, Set $>$: on b_1, i.e, event indicating the start of the triggering operation, a rule R_1 is triggered and stops after condition evaluation. All b_is that occur in this period, are consumed by the R_1. Once the event e_1 is received, rule R_1 resumes execution and completes its last phase $R_1(A)$. All e_is, whose complementary b_is have been consumed by the rule R_1. The next b_i that occurs, a rule is triggered and the same strategy is followed.

Conclusion. To sum up on our example, the rule execution dimensions of the thread management rule take the following values: *Event Processing Mode : set*, *Execution Mode : defered*, *Activity Mode : separate*. In other words, the rule is executed once for a set of events, after the triggering operation's execution returns, in a separate execution thread. In our simple scenario with only one rule and the three dimensions (including one with 3 values) that we consider in this article, we already get 12 combinations of values, i.e. potentially twelve different ways of executing the rule. Note that this does not have a big impact with our sample basic rule but think of a real system with many rules. To the rule programmer facing the complexity of rules semantics (execution model), we propose a architectural framework in which rule semantics is explicitely programmed/embodied into software components.

4. ARCHITECTURAL INTEGRATION OF ECA RULES IN COMPONENT-BASED SYSTEMS

As advocated by IBM in its autonomic computing manifesto [9], a *supervision loop* (or known as *control loop* in control theory terminology [11]) has to be realized in order to provide autonomic behaviour to a target system. On similar lines, we define an autonomic system as composed of an autonomic infrastructure superimposed on a target system, where the autonomic insfrastructure is responsible for implementing the control loop. At the heart of the control loop are reaction mechanisms that, on analysis of the events of interest, determine the action operations needed to achieve the objectives. Our reaction mechanism is formed of active rules, whose structure and execution have been explained thoroughly in the previous section. The thread management rule, defined in the introduction section, is one such active rule. This section details how such rules are architecturally integrated with an underlying target system.

If it is legitimate to work towards adding autonomic behaviour a posteriori to any system and even more to tackle explicitly existing legacy systems, we believe it is likely more advantageous to build explicitly, a priori, the system in a "certain way" to be able to make them autonomous in a flexible and generic way. This "certain way" is the component-based approach - and more precisely the Fractal component model [2], which has, according to us, interesting properties for the realization of autonomic systems. It has been mentioned before that our webserver - Comanche - is implemented as Fractal components.

4.1 Canonical Autonomic Architecture

An architecture for autonomic computing[14] must accomplish some fundamental goals, outlined in IBM's autonomic computing manifesto[9]:

1. It should possess knowledge about itself and about its execution environment in order to be able to detect modifications taking place externally in its environment, or in its behaviour to subsequently undertake corrective actions. It must describe how to compose these components so that the components can cooperate toward the goals of system-wide self-management.

2. It should be adaptable, i.e., its construction should be based on a structuring model which can isolate its constituting elements, and subject them to adaptations - and on operational techniques to actually perform these adaptations (interception, programs transformation, etc.). It should be able to dynamically adapt or reconfigure itself to varying and unpredictable environments without any explicit user intervention.

These key features are present in the Fractal component model, and we believe Fractal/Julia (its implementation in Java) is a suitable substrate framework for autonomic systems development as illustrated in the sequel.

Fractal Component Model. The Fractal[2] initiative aims at supporting component-based development, deployment and management (monitoring and dynamic reconfiguration) of complex software systems, including in particular operating systems and middleware. It includes several extensions coming from research works, for management (e.g. Fractal JMX), security, transactions support, etc. Fractal is also used for developing several middlewares such as Speedo - a Java Data Object implementation, CLIF - a load injection framework, etc[2]. The Fractal component model relies on some classical concepts in CBSE: *components* are runtime entities that conforms to the model, *interfaces* are the only interaction points between components that express dependencies between components in terms of *required/client* and *provided/server* interfaces, *bindings* are communication channels between component interfaces that can be primitive, i.e. local to an address space or composite, i.e. made of components and bindings for distribution or security purposes. Fractal also exhibits more original concepts. A component is the composition of a *membrane* and a *content*. The membrane exercices an *arbitrary reflexive control* over its content (including interception of messages, modification of message parameters, etc.). A membrane is composed of a set of *controllers* that may or may not export control interfaces accessible from outside the considered component. For runtime information on the component system, the control interfaces provide with (meta) information about the its structure and also means to manipulate this structure. The model is *recursive (hierarchical) with sharing* at arbitrary levels. The recursion stops with base components that have an empty content. Base components encapsulate entities in an underlying programming language. A component can be shared by multiple enclosing components. Finally, the model is programming *language independent* and *open*: everything (e.g. controllers, type system) is optional and extensible[3] in the model, which only defines some "standard" API for controlling bindings between components, the hierarchical structure of a component system or the components life-cycle (e.g. start, stop).

[2]CLIF, Speedo and other middleware engineered with Fractal are available in open source at http://www.objectweb.org/.

The Julia Implementation. Julia is an execution support for Fractal components written in Java. It is a full-fledged implementation of Fractal that supports the highest conformance level. More fundamentally, Julia is a software framework dedicated to components membrane programming. It is a small run-time library together with bytecode generators that relies on an AOP-like mechanism based on mixins and interceptors. A component membrane in Julia is basically a set of controller and interceptor objects. A mixin mechanism based on lexicographical conventions is used to compose controller classes. Julia comes with a library of mixins and interceptor classes, the component programmer can compose and/or extend. It is worth mentioning that Julia's membranes are particularly suited to insert sensors for observing and actuators for controlling components.

(Re)Configuration Languages. For the configuration and deployment of a Fractal-based system, an Architecture Description Language (ADL), known as Fractal ADL, is used to describe the system architecture. It is XML-based and strongly typed. It describes the interfaces of components (names and signatures), the subcomponents, the bindings between the various components, the initial values of component properties and the implementation of primitive components(e.g., the name of a Java class). All static information on a component is provided by the Fractal ADL. FScript is a scripting language used to describe architectural reconfigurations of Fractal components. FScript includes a special notation called FPath (loosely inspired by XPath) to query, i.e. navigate and select elements from Fractal architectures (components, interfaces...) according to some properties (e.g. which components are connected to this particular component? how many components are bound to this particular component?). FPath is used inside FScript to select the elements to reconfigure, but can be used by itself as a query language for Fractal.

Most of our approach relies on the Fractal component model. Of course, in realistic industrial settings, we cannot assume a whole distributed system to be Fractal-based; but we argue that those non-Fractal parts (legacy components) can be wrapped into Fractal components, and extended with autonomic behaviours. In the same line of thought, we believe that it would be very advantageous to carry out the development of the autonomic infrastructure itself in the form of Fractal components so as to consider the autonomic management of the autonomic infrastructure itself. (Our work

[3]This openness leads to the need for conformance levels and conformance test suites so as to compare distinct implementations of the model.

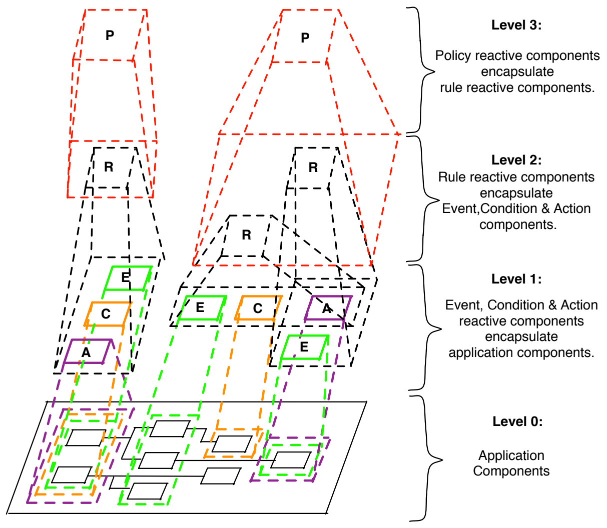
P
P
R
R
R
E
C
A
E
C
A
E
Level 3:
Policy reactive components encapsulate rule reactive components.
Level 2:
Rule reactive components encapsulate Event,Condition & Action components.
Level 1:
Event, Condition & Action reactive components encapsulate application components.
Level 0:
Application Components

Figure 3: Architectural Vision

can be considered as the first steps in this direction.)

4.2 Reactive part of the Canonical Autonomic Architecture

The architecture of the autonomic infrastructure is inspired from the fundamental management notion of *domain*[13], which consists in grouping the components on which the various reactive operations can be carried out. More formally, a domain is:

- *a unit of composition* to enable physical or logical partitioning of the application components, and
- *a unit of control* to define the type of control that needs to be carried out on these components.

The similarities between a Fractal component and the concept of domain suggest that a domain can be aptly modelised as a Fractal component. To incorporate reactive behaviour, several types of domains have been defined, each with a particular type of control unit applied onto its composition unit. They are each represented by a Fractal component, known as a *reactive* component. The autonomic infrastructure is formed by these reactive components. The various reactive components with their specific functionalities are listed below and illustrated by Figure 3.

- An *Event* (E) reactive component encapsulates application components where events of interest need to be detected. The membrane of this reactive component is responsible for identifying the application components that would belong to the content, instrumenting them appropriately. Further, on the occurrence of events of interest, they are notified to the appropriate controller inside the membrane of the component which processes them as defined in the rule execution model.
- A *Condition* (C) reactive component contains application components that represent the scope the queries that are to be evaluated. The functionnalities of its membrane include identifying the application components that would be in its jurisdiction, and evaluating queries on them.
- An *Action* (A) reactive component encapsulates application components, on which actions are executed. The type of control enforced by the Action component's membrane involves identifying its content's constituents, and executing some corrective operations on request.
- A *Rule* (R) reactive component coordinates the processing of a rule. It contains (exactly) one instance of the 3 above reactive components, i.e, Event, Condition (optional) and Action. The control applied by its membrane is the execution coordination of these reactive components. It is responsible for the execution of the rule embodying on a particular rule execution model.
- A *Policy* (P) reactive component's sole purpose is to coordinate the execution of the rules based on an execution strategy for a set of rules. The content part of the policy reactive component contains rule components, and its membrane controls the rights to their execution. Only, on explicit notification by the policy membrane, can the rules, once triggered, continue processing.

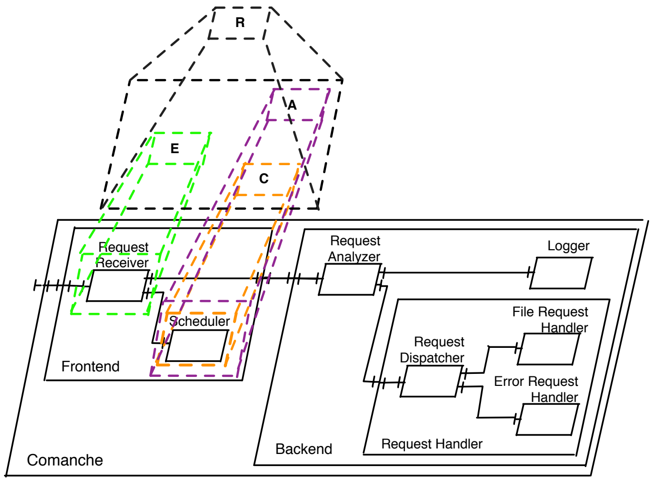

Figure 4: Thread Management Rule

Figure 3 shows the relationships between the various reactive components. The architecture employs key features of the Fractal Component Model [2], notably: the containment relationship - which can be found in components in the top three levels, where each component has components from the lower level as sub-components, e.g, the Policy component at level 3 contains the Rule components of level 2; and overlapping of reactive domains, thanks to the sharing property, e.g, an application component can belong to several reactive components. Figure 4 represents the implementation of our thread management rule. The Event reactive component encapsulates the request receiver component, because

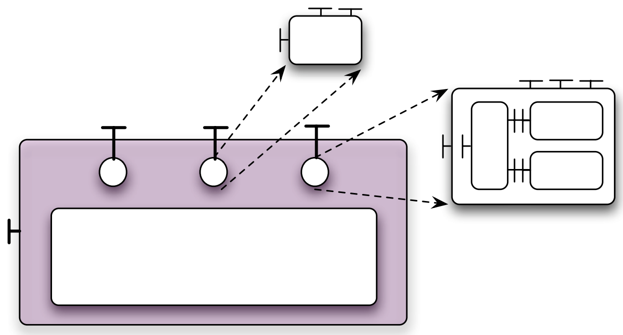

Figure 5: Extended Fractal Component

the event of interest for the rule occurs on it. The condition and action parts encapsulate the scheduler component. For their respective operations, that of verifying whether the thread pool is empty and increasing the size of the thread pool occurs on the same component. Finally, we have the Rule composite that encapsulates the reactive components (Event, Condition & Action) to coordinate their execution. The Policy reactive component is inexistant in the figure since, due to space limitations, only a single rule has been taken as an example to illustrate the framework.

5. AUTONOMIC INFRASTRUCTURE

This section presents the extensions in the prototypal implementation of our proposed framework, that offers the flexibility feature of our autonomic infrastructure.

Each of the reactive components presented above encapsulate either application components or other reactive components, where structurally both these types of components are similar. But the controller unit of each of these reactive components differs from one another because they implement different dimensions of the rule execution model. Therefore, the generic structure of a reactive component is a standard Fractal component - a composite component to be precise, with a flexible membrane formed by a set of newly defined and existing controllers (cf. Figure 5). The Fractal specification contains several examples of useful forms of controllers, which can be combined and extended to yield components with different reflective features. Likewise, additional controllers, not defined in the Fractal specification, can also be defined and incorporated in the membrane of a component. This permits the reactive components to have a membrane adapted to its respective rule execution dimensions. The membrane of a reactive component is composed of the following types of controllers : i) standard Fractal controllers as defined in the Fractal specification, ii) standard Fractal controllers represented as classical Fractal components with extended operations and finally iii) new controllers represented as Fractal components.

For instance, the membrane of the Event reactive component is composed of: i) an extended *attribute controller* to specify the parameters of the rule execution dimensions, ii) a *content controller* to add/remove the application components and iii) a *event processing controller* to process the events of interest and notify its enclosing rule component. The event processing controller is a newly defined controller represented as a Fractal component. It is a composite component with corresponding sub-components for the following dimensions: execution mode, activity mode and event processing mode. However, at the time our framework was developed, the set of controllers that constitute the membrane of a component could not be dynamically modified in the Julia implementation, nor could the functionalities of the existing controllers be modified. We have thus extended the Julia implementation to support these necessary features.

6. RELATED WORKS

Decision-making/reaction mechanisms form the core of an autonomic control loop, several systems use rules that specify conditions to be monitored and operations that should be executed when certain conditions are detected. *Production rules* (or *deductive rules*) have the following format - "IF *condition expression* THEN *action list*". These rules can also been extended, as in the DIOS++ framework[10], where an "ELSE" part as been added at the end - "IF *condition expression* THEN *action list* ELSE *action list*". ACEEL [3] uses *adaptation rules* which are couplets of the form "*OnEvent* : *Action*" with the first part defining the triggering event and the latter describing what actions to be performed. Inspired by triggers, we use active rules, which can be considered as a combinaison of the above two broad types of rules: "ON *event expression* IF *condition expression* THEN *action list*". Beyond these syntaxical differences, the main differences between production (or adaptation) rules and active rules concen their execution model. The execution model of production rules is based on the Rete algorithm: events are seen as facts which are added to the knowledge base; rules infer new facts from these facts ; the process stop when a fixed point is reached (no new facts can be inferred). Production or adaptation rule execution models are thus fixed and invariant. By contrast, active rules models are much more powerful and flexible. Also active rules models encompass the connection between the target system and the reaction mechanism while production rules systems do not (inference by production rules is disconnected from the actual execution of the target system).

For incorporating these active rules, the approach followed in SAFRAN[5], K-Components[7] consists in enhancing the computational component model with rule abstractions, where all rules concerning a particular component are injected into to it. Another approach, followed in Autonomia[6] or Automate[1], consists in implementing an autonomic computing infrastructure that acts as a control layer superimposed on the application, that provides the application as well as its individual components with the basic autonomic services to make it autonomic. In our approach, rules are not adhoc features injected into components but are themselves first-class components which can be manipulated as such. Thanks to the domain concept, the architectural connection between the application components and the rules are through containment relationships (hence a rule is not tied to a single component). Enabling thus, easy modification of the rule constituents (Event, Condition & Action).

In summary, to our knowledge, several works have tackled the architectural issues involved in the implementation of a control loop for autonomic features but none make such an explicit and extensive use of component programming for implementing the autonomic features themselves as in our proposition. As a consequence, these approaches often result in ad-hoc and not flexible management of the autonomic features. In most approaches that consider rules of a sort or another (deductive, active, etc.) as the core mechanism for autonomic features, rules execution issues have not been addressed in depth. Their execution strategy/methodology have been taken for granted, and many issues have been left under specified and ambiguous (we only find a brief mention of rule execution model in SAFRAN).

7. CONCLUSION

This article focuses on the architecture and behaviour of autonomic control loops. It proposes to use active rules as a decision-making mechanism, for which we have proposed i) a rule model, which is composed of a rule definition model and a rule execution model, to provide a clear semantics for the integration of rules, and ii) a flexible architecture that permits to dynamically add/delete new rules, to modify the rule definition model as well as the rule execution model of rules. Our rule execution model comprises of a set of dimensions, which we claim is not fully comprehensive, other dimensions can be envisaged. But we do claim that our generic architecture can incorporate new dimensions not yet identified. Further, our autonomic infrastructure takes into consideration the evolution of the target system. We would like to add that our work being positioned on components, i.e, our autonomic infrastructure as well as the underlying target system being component-based, has permitted us to benefit from CBSE properties.

The proposed autonomic infrastructure was developed in a Java implementation of the Fractal component model. The dimensions outlined in the article have been implemented, and experimented on the Fractal-based Comanche webserver. Some other execution dimensions, e.g. those related to the execution of multiple rules, which have not been discussed here due to space constraints have also been implemented.

Several future research directions are envisaged. In order to assess the validity of our proposition, we wish to apply it on more realistic applications. This would eventually permit us to determine the set of execution dimensions that are most relevant for component-based systems. Further, to enrich our proposition, we foresee a formalism for the definition of active rules that would typically be an extension of the Fractal ADL. This extension should be quite straitforward, for Fractal ADL is actually modular and extensible. On a longer term, we shall study the problem of interference between the behaviour of a control loop (i.e. the rules in our case) and the target system and the stability of the global system (target system and rules).

Acknowledgments. This work is partially supported by the French RNTL Selfware project and the European IST Selfman project. The authors thank A. Lefebvre and B. Dillenseger for their careful reading and comments.

8. ADDITIONAL AUTHORS

9. REFERENCES

[1] M. Agarwal, V. Bhat, H. Liu, V. Matossian, V. Putty, C. Schmidt, G. Zhang, L. Zhen, M. Parashar, B. Khargharia, and S. Hariri. Automate: Enabling autonomic applications on the grid. *Autonomic Computing Workshop*, pages 48–57, 2003.

[2] E. Bruneton, T. Coupaye, M. Leclercq, V. Quema, and J.-B. Stefani. The FRACTAL component model and its support in Java: Experiences with Auto-adaptive and Reconfigurable Systems. *Softw. Pract. Exper.*, 36(11-12):1257–1284, 2006.

[3] D. Chefrour. Developing component based adaptive applications in mobile environments. In *SAC '05: Proc of the 2005 ACM symposium on Applied computing*, pages 1146–1150, New York, NY, USA, 2005. ACM Press.

[4] T. Coupaye and C. Collet. Detailed sketch of a parametric execution model for active database systems. Technical report, LSR - IMAG Laboratory, University of Grenoble. France, 1997.

[5] P.-C. David and T. Ledoux. An Aspect-Oriented Approach for Developing Self-Adaptive Fractal Components. In *5th International Symposium on Software Composition (SC'06)*, Lecture Notes in Computer Science, Vienna, Austria, march 2006. Springer-Verlag.

[6] X. Dong, S. Hariri, L. Xue, H. Chen, M. Zhang, S. Pavuluri, and S. Rao. Autonomia: an autonomic computing environment. In *Proc of the 2003 IEEE Int'l Conf on Performance, Computing, and Communications*, pages 61–68, 2003.

[7] J. Dowling and V. Cahill. The k-component architecture meta-model for self-adaptive software. In *Proceedings of the Third Int'l Conf on Metalevel Architectures and Separation of Crosscutting Concerns*, pages 81–88, London, UK, 2001. Springer-Verlag.

[8] S. Gatziu, A. Koschel, G. von Bültzingsloewen, and H. Fritschi. Unbundling active functionality. *SIGMOD Record*, 27(1), Mar. 1998.

[9] P. Horn. Autonomic computing: Ibm's perspective on the state of information technology. Technical report, IBM Corporations, October 2001.

[10] H. Liu and M. Parashar. Dios++: A framework for rule-based autonomic management of distributed scientific applications. In *Euro-Par*, pages 66–73, 2003.

[11] M. Kokar and K. Baclawski and Y. Eracar. Control Theory Based Foundations of Self Controlling Software. *IEEE Intelligent Systems*, 14(3):37–45, 1999.

[12] N. W. Paton, F. Schneider, and D. Gries, editors. *Active Rules in Database Systems.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1998.

[13] M. Sloman and K. Twidle. Domains: a framework for structuring management policy. *Network and distributed systems management*, pages 433–453, 1994.

[14] S. R. White, J. E. Hanson, I. Whalley, D. M. Chess, and J. O. Kephart. An Architectural Approach to Autonomic Computing. In *Int'l Conf in Autonomic Computing*, pages 2–9, New York, NY, 2004.

G Autonomic Performance Characterization of Components

Performance characterization of black boxes with self-controlled load injection for simulation-based sizing

Ahmed Harbaoui*
France Telecom R&D
Grenoble, France

Bruno Dillenseger†
France Telecom R&D
Grenoble, France

Jean-Marc Vincent‡
Laboratoire LIG, Projet MESCAL
Grenoble, France

Abstract **Sizing and capacity planning are key issues that must be addressed by anyone wanting to ensure a distributed system will sustain an expected workload. Solutions typically consist in either benchmarking, or modelling and simulating the target system. However, full-scale benchmarking may be too costly and almost impossible, while the granularity of modelling is often limited by the huge complexity and the lack of information about the system. We propose a methodology that combines both solutions by first identifying a middle-grain model made of interconnected black boxes, and then to separately characterize the performance and resource consumption of these black boxes. We also propose a component-based supporting architecture, introducing control theory issues in a general approach to autonomic computing infrastructures.**

I. INTRODUCTION

"Many organisations expensively invest to build distributed systems applications and web services and pay a huge amount of money to maintain and keep the environment up-to-date. In most cases, the overall capacity planning and procurement is done without a defined methodology"[7].

This kind of situation is responsible for important loss of incomes, ranging from losing customers on an on-line purchase service to losing stock exchange transactions. Hence, it shows the utility of an infrastructure's capacity planning to support the associated load. In this context, our work comes from the problem of planning capacity of a distributed infrastructure to support a given load. While simulation techniques are developed in order to predict the performances, and to detect the bottlenecks and critical resources, the preliminary modelling phase of the system typically encounters opacity problems when a certain level of granularity is reached. Then, "Black boxes" appear, either because of a lack of information about their behaviour, or because of their great complexity. However, the modelling of the global system is impossible without a minimal model of these black boxes, including resources consumption. In this paper we deal with the problem of modelling parts of the system as black boxes. Some works studied methods for black boxes characterization. [6], [9] use an analytical model by considering the whole system as a one black box. We start in section 2 a discussion on the different approaches concerning the estimation and the determination of performance models. Then, we present the approach enabling the determination of parameters influencing our system. In the third section, we present the CLIF framework and this integration to our approach. Section 4 deals with problems of stability and saturation. The next section experiments our approach with a simple example. Finally, we give some ideas to study in future work.

II. MODELS DISCUSSION

Our goal is to generate black boxes models. These black boxes result from a lack of information concerning the behaviour and resource consumption, or a high level of complexity of some parts of the global system. Then, the generated models will be integrated in the global system model. With regard to this modelling problem, several approaches may be adapted. To begin, we present these approaches:

- *analytical modelling* consists in reducing the system in a mathematical model and analyzing it numerically. Several mathematical tools enable such a modelling: automata, Petri nets, probability approach (queuing network), etc.;
- *simulation* consists in establishing a simplified model for the system by using suitable software. This technique is commonly used to evaluate performance;
- with *traffic emulation*, direct measurements and analysis are carried out on the system. It gives a better understanding of the system's real behaviour. This kind of modelling does not need detailed information about the system. The model is generally built only by considering the outputs versus the inputs.

The software systems we want to qualify are distributed and complex. In general, they suffer from a lack of information describing their behaviours and interactions with their environment. In addition, we cannot access their source code. All these reasons make direct modelling a hard and complex task and lead us to adopt a traffic emulation approach since it does not require such information.

III. METHODOLOGY

The traffic emulation approach gives the performance model by considering the system output as a function of the input load. Load is injected in the system in order to qualify its capacities and to extract performances and resources consumption before saturation.

A. Defining Black Boxes

This part consists in identifying the black boxes of a system. Depending on the system, one tries to divide it into as many black boxes as possible. When decomposition becomes too complex, the system must be kept fully. Otherwise, we define mutual interactions among the black boxes and other parts of system. In fact, interactions could be external invocations of other black boxes, system calls, access to resources, etc.

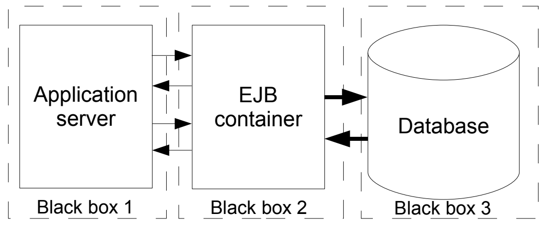

Figure 1. Example of J2EE application

Let us take the example of a J2EE web application, composed of an application server, an EJB container and a database. In such an architecture, an intuitive decomposition is possible that splits the system into three black boxes (see figure 1). The first one is dedicated to the application server, the second to the EJB container and the third to the database. This way, we obtain a more detailed and precise performance model.

B. Choosing the Performance Parameters

The parameters are the different characteristics that impact system performance. They depend on the type of target system and fixed goals. If we take the previous example - J2EE application - parameters could be: end-to-end response time, throughput in requests per second, number of customers per time unit, etc.

Given the important number of parameters that could influence the system performance and the huge amount of time needed for performances study, it looks more suitable to consider only relevant parameters which are directly linked to the aim of the study. If the choice looks difficult, a "factorial analysis" will enable to identify the actually important factors, through some experiments. In our J2EE example, we chose response time as the interesting performance factor.

C. Defining Workload and Instrumenting

Once the black boxes are identified, we define the load to apply through several uses cases and we execute the test. In our case (J2EE application), the load is defined through a number of typical usages consisting of interlaced sequences of requests and think times, and a parallel execution of a number of virtual users performing those usages.

However, since we want a good qualification of both the black box and the global model, it's necessary to apply a load that is as close as possible to the real load. In order to reach this goal, the testing platform may repeatedly replay pieces of real execution traces. Instrumentation deals with monitoring and measuring the use of resources (CPU, memory allocation and network occupation) by placing some probes in different parts of the system under test.

D. Modelling

Once we have collected performance measures associated to the applied loads, we will extract performance model based on these results. In order to model the system with queuing networks, we model each black box with a queue. Each queue is labeled by the performance characterization obtained in previous step. These queues could be represented in three different ways depending on the type of the black box. With *load-dependent resources*, queuing and service times depend on the load D.

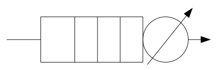

Figure 2. Queue for a load-dependent resource

The two other queue models are just particular cases of this model: *load-independent resources* represent resources where the service time does not depend on the load; *delay resources*' service time does not depend on the load and there is no queuing.

We have to identify the type of each black box (load-independent, load-dependent, etc.) according to the test results. The load test is executed on each black box. If our system is composed of several interacting black boxes, we define software-*plugs*. They replace interactions of the tested black box with other black boxes while conserving a constant value for performance parameters of interest. Then, one subtracts this constant from the value obtained from the test and hence we get the black box characterization. Let us return to our example of the J2EE application, to characterize black box 1 which interacts with box 2. One develops a software-plug that replaces box 2 with constant response times for each invocation. At the end of the tests, one withdraws software-plug constant from the global response time to obtain the first black box one.

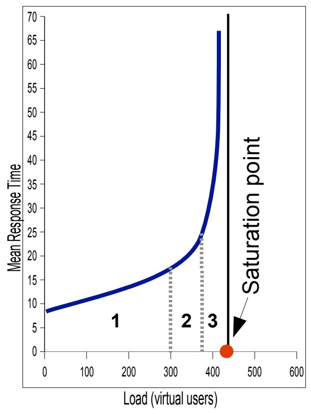

Figure 3. saturation point

After carrying out all tests, we draw response time as a function of the applied load. The result, as we expect, should be close to the one sketched in figure 3. In portion 1 in figure 3, response time linearly grows with load, which is a correct behavior for resource-shared processing. In portion 2, we observe the beginning of the effect of application contention. Approaching the saturation point, the system does not follow the imposed load any more, and its response time tends to infinity.

IV. SATURATION AND STABILITY OF SYSTEMS

All measurements should be done when the system reaches the limit just before saturation. However, if we wish to reach saturation, load injection should be done in such a manner that enables to be more and more close to this situation. First of all, one injects a minimum load and waits for the system to become stable. Then, one progressively increases the load to a higher level, waits for stability, and so on (see figure 5). This method could take a huge time depending on the system. This is why we propose in section V an infrastructure to automatically find the saturation point. To achieve this, the load injector is controlled through a feedback loop that observes the system response to the current load and makes the decision to increase or decrease the load with reference to the measured performances (cf. figure 4).

When looking for the saturation point, we must ensure that the system is stable during all the load ramp-up in order to get reliable and accurate results. The system is stable if its performance remains the same whenever the workload

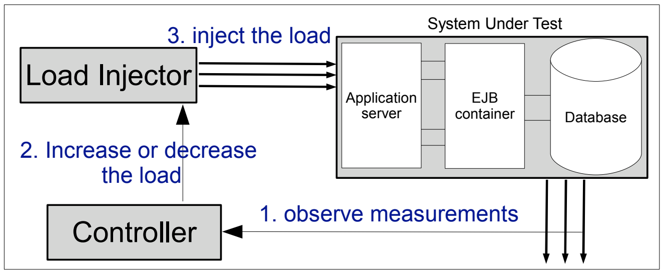

Figure 4. Load injection feedback loop

keeps the same. If the load ramp-up is too steep, it may be difficult to clearly identify the unstability area corresponding to the saturation point. For this reason, we have to maintain a constant load during a sufficient duration for the system to reach a stable state. Then, the duration as well as the load level for the following step depend on the response of the system to the current load level.

Stability criteria depend on the kind of system and the quality of service that must be provided. These criteria must be defined at the very beginning, just like the global performance parameters of interest. For example, in the case of a J2EE application, we may choose the maximum variation of response time as a stability criterion.

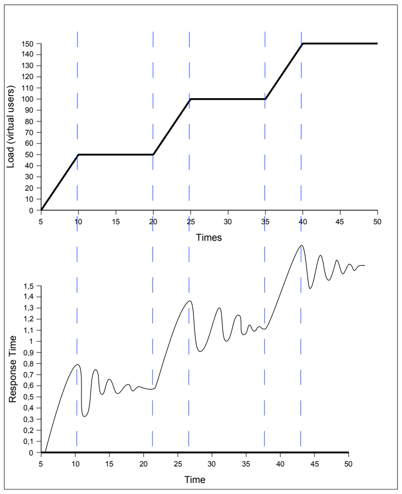

Figure 5. typical response time evolution under a step-by-step workload

The first graph of figure 5 shows the progressive level of load used to reach saturation. The applied load is a step-by-step function that enables to wait for stability once the load has been increased. If the stability condition is satisfied, we proceed with the next step (higher level load). Otherwise, we decrease the load until we obtain a stable situation. The second graph illustrates this load injection policy through a response time-based stability criterion. It sketches the variation of the system's response time as a function of the applied load. The system clearly reacts to the different load levels with an increase of the average response time and oscillations around the average that decrease while the load remains constant. When the response time oscillations are small enough to match the stability condition, we increase the load once again. This is repeated until the saturation point is reached.

V A component-based supporting infrastructure

A An architectural approach to autonomic computing

In order to experiment our methodology, we propose a practical software infrastructure that fits, on the one hand, genericity (our approach may be applied to any kind of black box), and, on the other hand, autonomy (self-regulated load injection). This is the reason why this work is carried out in the context of architectural research on autonomic computing. This approach has been proposed in [3], and is currently being developed in collaborative projects Selfware [2] and Selfman [8, 2].

As presented in [5], the basic idea of autonomic computing may be summarized as the principle of using computing power to automatically (autonomously) manage computing systems complexity. Our architectural approach to autonomic computing consists in relying on a uniform component-based representation of the target computing system, either in a native manner or a wrapper-based manner. Then, a feedback loop is introduced, with sensors at one end (observation), actuators at the other end (reaction/feedback control), and a decision element in between. The feedback loop relies on a communication middleware to handle observation events coming from the sensors, as well as reaction events coming from the decision elements to the actuators. More than just a plain transport service, this event middleware may also support aggregation, filtering and a variety of message delivery models (e.g. publish/subscribe, group communication). All elements in this architecture are uniformly represented and handled as components, using the Fractal component model [1].

B CLIF Load Injection Framework

Starting from this component-based and feedback loop-based architectural approach, we need to build a self-regulated load injection system. We need components that generate a workload on the System Under Test (*SUT*), and components that give feedback information about the resulting SUT performance (response time, throughput) and computing resource usage. Moreover, there should be a decision component that closes the feedback loop between observation and reaction, in order to dynamically and autonomously adapt the generated workload.

CLIF [4] provides a framework of Fractal components that meets these requirements. Main components are: *load injectors* for traffic generation and response times measurement, *probes* for monitoring the consumption of arbitrary computing resources, and a *supervisor* component which is bound to all injectors and probes and provides a central point of control and monitoring. While the typical CLIF usage consists in plugging a user interface on the supervisor, we are simply going to bind an autonomic *controller* component to the supervisor and discard the user interface. This is actually done by developing this controller component and slightly modifying an XML file describing the CLIF application, using a commonly called Architecture Description Language. As shown by figure 6, this controller component is bound to other components:

- a load injection policy component that computes the control feedback on the load injection system;
- a saturation policy component, that detects whether the SUT is saturated or not.

that computes the control feedback on the load injection system according to the observation of response times, resource usage and possible alarms. Both components rely on the observation of response times, resource usage and possible alarms.

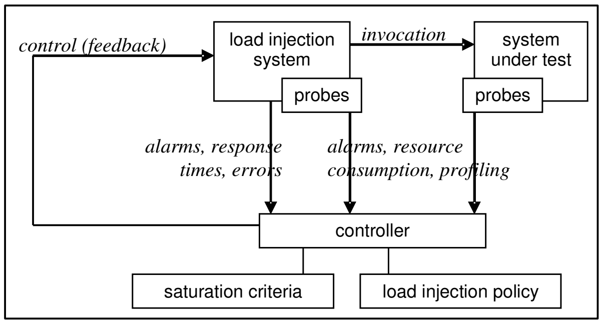

Figure 6. A CLIF assembly for self-regulated load injection

In order to vary the load level during the saturation look-up process, we use the classical virtual user concept supported by CLIF. A virtual user is a computer program that invokes the SUT in a similar way that a real user would do. Load testing consists in massively and concurrently running virtual users. Each CLIF load injector is actually an execution engine for such virtual users. Then, the workload regulation performed by the controller component simply consists in adjusting the number of virtual users run by the load injectors according to the observation. Here, it must be underlined that the load injection policy can be generic, since it may only handle the concept of virtual user whatever the actual SUT is. Pure control theory-based algorithms may apply there. As far as the saturation policy is concerned, it may be defined in a generic manner also, but it may be chosen or parameterized in adequacy with the SUT. Simple, generic saturation detectors are: response time threshold, error or alarm occurrence, or request throughput stagnation.

VI An experiment

A *Rationale*

We propose a self-regulated load injection experiment based on our component-based architectural approach to autonomic computing, using the Fractal model and CLIF load testing framework. The target system under test is an Enterprise Service Bus (*ESB*), a kind of request broker used in Service Oriented Architectures to support mediation features such as accounting, routing, logging, security, management of service level agreement, etc. This ESB is the black box we want to characterize from the performance point of view. The system clients are emulated by virtual users running in CLIF load injectors and generating SOAP requests. Real services are replaced by software plugs, i.e. dummy services that reply to requests with a constant response time, whatever the incoming workload. Of course, the plugs' performance must be qualified before, to determine this response time and the correct operating range with regard to the incoming traffic throughput.

With this simple experiment, we are just going to show how the looped load injection system is going to find the ESB saturation limit, in terms of maximum sustainable number of virtual users and request throughput, according to a given saturation criteria. The behavior of our virtual users consists in generating 20 requests during 20 seconds before exiting, with random think times between consecutive requests, which gives an average of 1 request per second per virtual user. The ESB is based on a dedicated hardware platform, which offers load percentage information through the SNMP protocol. We have defined a new CLIF probe to get this information.

The controller starts with one virtual user per load injector. Then, it proceeds through 20 seconds iterations, observing the ESB's average load percentage, comparing it to a given threshold (80% here), and deciding a new number of virtual users: increase that number when the threshold is passed, decrease when it is unreached. We see that we actually implement a control feedback function, with all the associated issues in terms of stability and reactiveness. This control feedback is rendered by the load injection and saturation policy, provided as simple algorithmic rules here, but this may be easily replaced in the architecture by arbitrarily complex and advanced computations relying on the observations from the load injectors and probes. For instance, the iterations duration shall not be constant but computed at runtime. Of course, more probes would be necessary, in the general case, not only for the sake of saturation lookup, but also to go further towards our final goal of full characterization for system simulation and sizing.

B *Results*

The results presented below have been produced with 4 load injectors and a controller distributed on 5 distinct computers (Intel bi-Xeon or AMD bi-Opteron, 2 or 3 GB RAM, Gb/s Ethernet, Linux kernel 2.6.15-1-686-smp). The ESB load probe is hosted on a 6th computer and simply gets information from the ESB platform's SNMP agent. The observation (see figure 7) shows promising results, particularly because this ESB platform had already been "manually" benchmarked with CLIF's common user interface on the same infrastructure, giving similar results. After 3-4 minutes, we see a rather quick and good stabilization of the number of virtual users around 400 and an ESB load around 80%. As expected, the request throughput is roughly following the number of virtual users (just a little smaller), with some sudden drops at time 270s and 390s, that can be explained by the occurrence of garbage collector on the load injectors. To be more accurate about this phenomenon, we should add CLIF's probes on the load injectors, and especially the JVM probe which detects occurrences of garbage collection. Garbage collection is the typical kind of phenomenon that must be taken into account to prevent unstability problems.

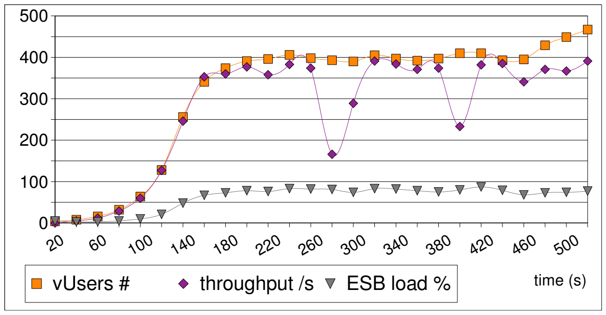

Figure 7. Automatic saturation of an ESB platform

VII. EXTENSIONS

This work is being applied to current R&D projects in France Telecom, where the characterization of black boxes performance and resource consumption is key to develop and to keep good working conditions for many infrastructures. Sizing and capacity planning are essential.

For example, in the case of Machine to Machine (M2M) services, a large number of machines (teller machines, detectors, cameras, boilers, etc.) exchange events and a variety of data. The M2M middleware also controls its own execution by observing resources usage. Such infrastructures are typically overlay networks, that are widely distributed, generate huge amount of events and connect a great number of devices together. Breakdowns may be frequent in such systems, and the manual supervision and management of such big infrastructures is almost impossible. Here, autonomic computing (see section A) research becomes fundamental to support self-optimization, self-healing or self-configuration features.

An M2M overlay network is basically a set of nodes, performing arbitrary computations that produce events, connected together through an arbitrary network topology. The nodes typically perform arbitrary business computations that are unknown to the network operator. As a consequence, the queuing model and our approach applies quite well to the global M2M system, where nodes are black boxes. The nodes must be tested one by one with our self-regulated load injection platform in order to produce the necessary performance characterization. Then, we will be able to simulate the global system and provide M2M systems with support for sizing and capacity planning. Moreover, in the context of autonomic computing, it will be possible to evaluate self-reconfiguration decisions through simulation, before actually performing them, to prevent unexpected performance defects.

VIII. RELATED WORK

Two previous works in black boxes modelling could be used as references in this work. The first deals with black boxes modelling in a particular context which is storage environment and the second proposes a method to determine relevant and necessary parameters to estimate a performance model of black boxes.

In [9], the authors evaluate the most popular techniques used in black box modelling in storage environment and measure the precision of each technique to obtain the best of them. [6] tries to determine necessary properties to estimate performance model for black box when it is used in a feedback loop.

These papers use an analytical model by considering the system as a one black box unlike our method which decomposes the system in several black boxes and hence gives a more detailed model. If we have to do a factorial analysis to determine relevant parameters, we can use results of the second article which proved that the method of least squares does not give the best estimation any more when a control loop is used. Furthermore, a regression method can not be used since we are looking for performance before saturation which means we are not in the linear range.

IX. CONCLUSIONS AND FUTURE WORK

In this paper, we addressed the general issue of sizing and capacity planning of distributed systems, by proposing a combination of global system modelling and real testing of small, unknown elements (black boxes). The proposed methodology consists in characterizing the performance and computing resource consumption of the black boxes, by generating a variable workload on them and observing their behaviour, and to use these results as an input in the global system model. Then, this model will be used to predict the adequate sizing of the execution support as well as the expected performance. To achieve this prediction, we chose a queuing network model and a simulation-based approach.

We also presented a component-based software architecture to support the autonomous characterization of black boxes. Springing from architectural research for autonomic computing infrastructures, it relies on a load injection framework with a feedback control loop. We partly implemented end experimented this architecture in a real test case with an Enterprise Service Bus. The promising first results still require more research work in several directions, such as: identifying the black boxes, factorial analysis, saturation and stability criteria, control theory, and of course simulation to achieve our ultimate goal in terms of sizing and capacity planning. Our future work will be guided by this goal, in the context of Machine-to-Machine applications and related middleware.

References

[1] E. Bruneton and al. The fractal component model and its support in java. *Software Practice and Experience, special issue on Experiences with Auto-adaptive and Reconfigurable Systems*, 36(11-12), 2006.

[2] S. F. R. collaborative project. Déploiement, configuration et administration autonome de systmes rpartis. http://www.rntl.org/projet/resume2005/selfware.htm, 2005.

[3] T. Coupaye, F. Horn, and al. Principes généraux darchitecture logicielle pour la construction dapplications autonomiques ouvertes. *L'autonomie dans les réseaux (Traité IC2 série Réseaux et télécommunications)*, pages 1–34, September 2006.

[4] B. Dillenseger. Flexible, easy and powerful load injection with clif version 1.1. In *Fifth Annual ObjectWeb Conference*, http://objectwebcon06.objectweb.org/xwiki/bin/Main/DetailedSession, January 2006. ObjectWeb.

[5] IBM. An architectural blueprint for autonomic computing. http://www-03.ibm.com/autonomic/pdfs/AC Blueprint White Paper V7.pdf, June 2005.

[6] M. Karlsson and M. Covell. Dynamic black-box performance model estimation for self tuning regulators.

[7] D. A. Menasc and V. A. F. Almeida. *capacity planning for web services: Metrics, Models and Methods.* 2002.

[8] P. V. Roy and al. Self management of large-scale distributed systems by combining peer-to-peer networks and components. Technical Report TR-0018, CoreGRID, December 2005.

[9] L. Yin, S. Uttamchandani, and R. Katz. An empirical of black-box performance models for storage systems. *14th IEEE International Symposium on Modelling, Analysis and simulation of computer and Tlcommunication Systems (MASCOTS '06)*, October 2006.