H-0219 (H0403-002) March 27, 2004
Computer Science

IBM Research Report

The Generic Manageability Library (GeMaL)

Y. Aridor, Y. Gal, Z. Har'El, A. Orlovsky, B. Rochwerger, M. Silberstein
IBM Research Division
Haifa Research Laboratory
Haifa 31905
Israel

The Generic Manageability Library (GeMaL)

Y. Aridor, Y. Gal, Z. Har'El, A. Orlovsky, B. Rochwerger, M. Silberstein
IBM Haifa Research Lab.
{aridor,yoavg,zharel,arieo,rochwer,marks}@il.ibm.com

Abstract

The OGSA Generic Manageability Library consists of a set of Grid Services definitions and their Java implementation aimed at simplifying the development of self-managing systems according to the IBM autonomic computing architecture [][1].

GeMaL defines a generic manageability interface that developers can use to wrap different components and manage them in a standard fashion. GeMaL is defined on top of the Open Grid Services Infrastructure (OGSI) [12] and few additional operations to fulfill some functionality gaps. These are discussed in detail in this paper. Moreover, GeMaL supports hierarchical (multi-layered) management systems by supporting composability (of a multi-layered management system), pluggability (of different management tools e.g., analyzers) and configuration (of the management system). Finally, GeMaL is composed of a very limited set of interfaces which are easy to use in autonomic systems.

Several components have already been GeMaLized (wrapped with GeMaL interfaces) and are included with the library: Tivoli TAME, the IBM AC Generic Adapter (GA), IBM Solution Install (SI), Apache, Xindice, and Tomcat. These components are managed, via GeMaL interfaces, (autonomically) by the Tivoli TAME tool or (manually) by the GeMaL Visual Observer - a graphical management tool included with the library.

1 Introduction

The IBM Autonomic Computing Architecture [1] is about building self-managing systems to reduce the increasing complexity of managing IT systems. The basic principle behind this architecture is that self managing systems are implemented with an intelligent control loop. This control loop is composed of two elements: a *managed element* and an *autonomic manager*. As shown in Figure 1, the managed element (ME) is the controlled system component. The autonomic manager (AM) is the component that implements the control loop. The autonomic manager gathers information from the managed element through its *sensors* and changes the state of the managed element through its *effectors*. The combination of the sensors and effectors form the manageability interface. The control loop itself is divided into four functional parts: Monitor, Analyze, Plan and Execute; these communicate through asynchronous messaging and share common data in the Knowledge component.

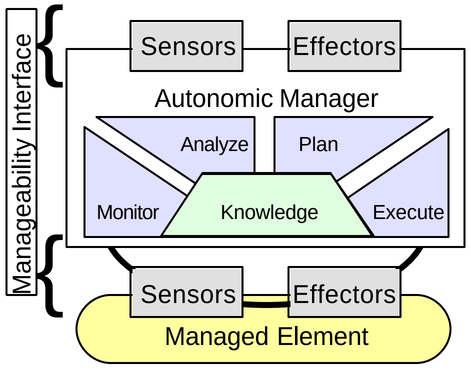

Figure 1: The autonomic computing components and the MAPE-K control loop.

Consider now a typical multi-tier application. There are several possibilities on how to build autonomic mangers to control complex environments like this. For example we can build a flat autonomic manager controlling the entire site (Figure 2), in which case it will need to deal with issues such as collecting and understanding data from multiple sources, and issuing commands to the different components. Alternatively, we can build a hierarchical autonomic manager (Figure 3), with tier-specific autonomic managers and higher level manager which will need to deal with the coordination among the different managers. Clearly, there are many other possible combinations and the answer as to which approach is the best is not clear and it may be from different solutions.

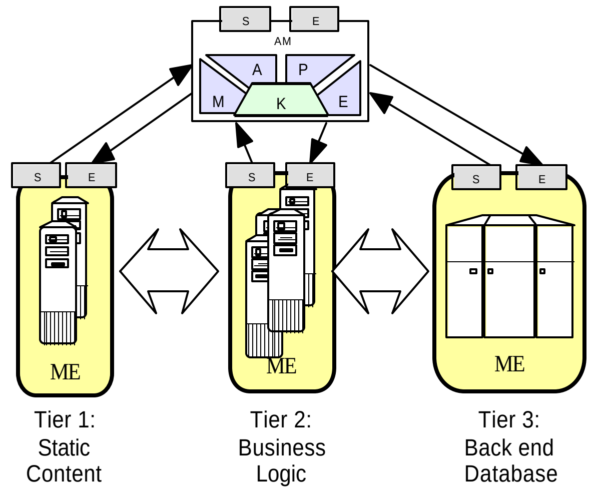

Figure 2: Flat autonomic management of a typical multi-tier application:

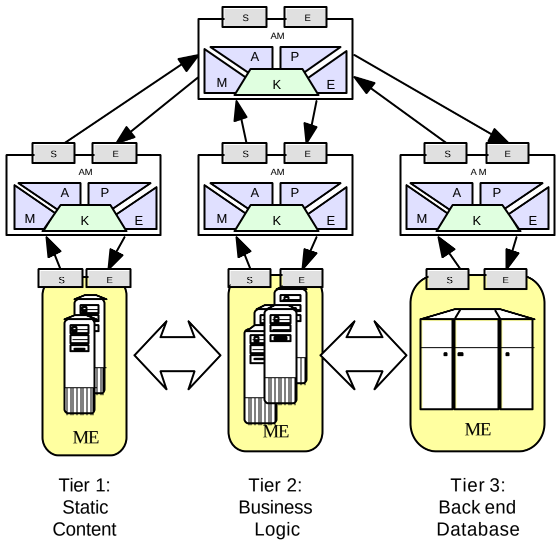

Figure 3: Hierarchical autonomic management of a typical multi-tier application

In environments like this, it is desirable to remove from the autonomic managers the need to deal with details specific to a particular managed element (data format, available commands, etc.). Furthermore, complexes like these are built today out of non autonomic components, hence a mechanism to wrap existing components/system with autonomic computing interfaces is required. These are the issues that the Generic Manageability Library (GeMaL) presented in this paper deals with: how do we virtualize managed elements in a way that simplifies the task of building autonomic managers for heterogeneous distributed systems? Once we have this virtualization, how do we simplify wrapping up existing components so they easily can be plug into an autonomous complex system?

The Open Grid Services Architecture (OGSA) [5] has been proposed as a framework for the integration and management of distributed systems. In the spirit of OGSA, GeMaL virtualizes the different managed elements by defining a generic manageability interface and exposing it as Grid Services. Through this interface users of GeMaL cam compose hierarchical autonomic systems (composability), where each component can be replaced by another component with similar functionality (pluggability) and actions at all levels are coordinated among different autonomic managers (coordinated execution).

2 The Generic Manageability Library

Following the OGSA paradigm, we define the autonomic computing manageability interface as a set of OGSA services. The Generic Manageability Library (GeMaL) provides an extensible implementation of these services which simplifies the development of managed elements. To allow maximum flexibility, GeMaL classes are built as OGSA Operation Providers [7], one for each porttype defined. Using this approach, allows us to easily add the GeMaL functionality to any Grid Service.

2.1 The `Sensor` PortType

In the AC architecture data is collected from each managed element through the following two interactions styles:

1. In the *Retrieve-state* interaction style, the autonomic manager queries the managed element for a particular piece of information For state data e.g., CPU load. this can be mapped to the `findServiceData` operation of the `GridService` porttype. However, there are types of information which do not fit well into the *state data* definition. As shown in Figure 4, the GeMaL `Sensor` porttype provides additional operations to collect and query *history-like data*, i.e., data that is accumulated during the entire lifetime of the managed element such as logging information. The `queryEvents` operation is used to selectively (using XPath) retrieve data from the managed element history of events. Internally, applications can store this kind of history data in different places and format, hence we need to specifically tell the

managed element which data source should be exposed to the `queryEvents` operation; this is what the `collectEvents` and `stopCollectingEvents` operations are for.

2. In the *Receive-Notification* interaction style, the autonomic manager expresses interest in some data (by XPath based subscription); when matching data appears, the managed element asynchronously notifies the autonomic manager. Again, for state data, this could be mapped to the `subscribe` operation of the `NotificationSource` and the `deliverNotification` operation of the `NotificationSink` porttype. Notifications for history-like are necessary only for newly arrived events – the `Sensor` porttype defines the `LastEvents` service data element as a window of configurable size into the events history.

2.1.1 The `SensorProvider` class

GeMaL's default implementation for the sensor porttype extends the `NotificationSourceProvider` class included with the Globus Toolkit 3. It adds support for XPath based subscriptions[1], i.e., notifications are sent to each subscriber only when the changed service data element matches the XPath expression associated with the subscription. All implementations of the `Sensor` porttype should extend this class.

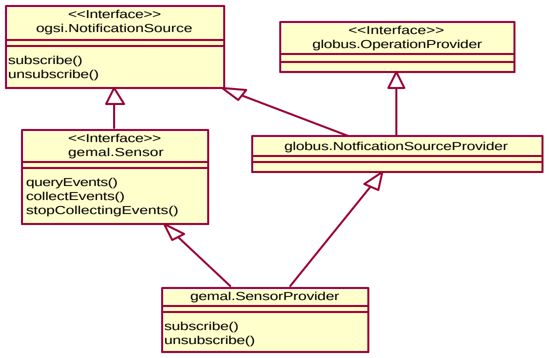

Figure 4: The `SensorProvider` class diagram. This class implements the `Sensor` interface and adds to the default `NotificationSourceProvider` support for XPath based subscriptions.

[1] A similar mechanism is available in the ogsa_messaging_jms.gar additional package for GT 3.0.2. Use of this package requires a JMS provider; we chose to build this feature into our provider to remove this dependency.

2.1.2 The `GenericSensorProvider` class

Although many types of data may exhibit the characteristics of history-like data, the focus in GeMaL has been logging data. When application specific logging data is to be exposed through the `Sensor` interface we need to worry about the translation of this data into a common format so autonomic managers can easily analyze data coming from different sources.

IBM's Generic Log Adapter [6] is a rule based tool that transforms software log events into the common base event (CBE) format [8]. This format, which has been submitted to OASIS for standardization, defines the structure of an event in a consistent and a common format.

In GeMaL we wrap the GLA under the `GenericSensorProvider` class: it spawns the GLA engine and on each call to the `collectEvents` method it adds a new data source with the corresponding configuration rules (a context in GLA terminology). GeMaL includes a customized GLA sink (the component at the end of the GLA processing chain), that catches the generated CBEs and updates the `LastEvents` service data element. The default GLA sink, which writes to a file, is also used; then, the `queryEvents` method is applied against the data generated by the GLA (see Figure 5).

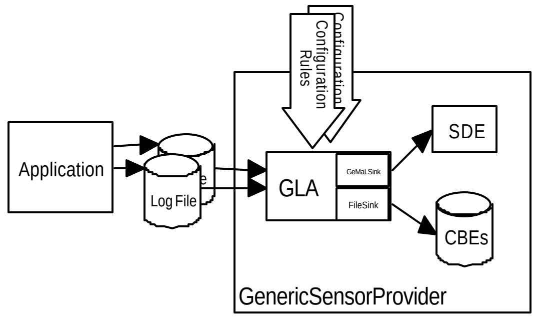

Figure 5: The `GenericSensorProvider` contains an instance of the Generic Log Adapter (GLA) to translate, with the use of user supplied rules, the application specific log file(s) a file with common based events. To allow notifications, the last CBEs generated are kept in a service data element by the GeMaLSink.

Clearly, the `GenericSensorProvider` class extends the `SensorProvider` class to support subscriptions to changes in the `LastEvents` service data element (see Figure 6).

The recommended approach to creating a sensor is to use the `GenericSensorProvider.` Alternatively, the `SensorProvider` can be extended to either optimize the conversion process or to produce a different data format.

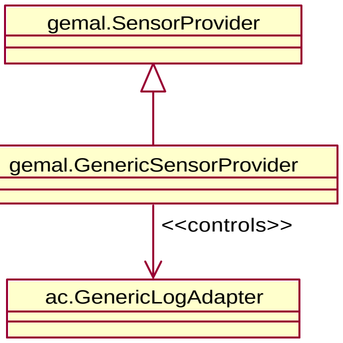

Figure 6: The `GenericSensorProvider` class diagram.

2.2 The `Effector` PortType

This porttype follows the *Perform-Action* interaction style, in which, the autonomic manager explicitly invokes, synchronously, an action on the managed element, e.g., reconfiguration. In a heterogeneous distributed system, built from many different components, the number of possible actions may be too high, and so will be the complexity of an autonomic manager for such an environment. Fortunately, even with a small set of well defined actions common across all components we can achieve simpler yet useful autonomic managers.
A good example of achieving manageability of diverse components through a generic interface is the System V `init` scripts [9] found today in many Linux distributions. Through a very small set of operations (`start/stop/restart/status`) the system is able to control a set of diverse background processes. To enable an application to participate in the `init` process a script is provided that maps each of the common operations into an application specific action.
GeMaL tries to extend this mechanism to autonomic systems by defining a small set of common actions that each managed element must implement. Managed elements can support additional actions, but to simplify interactions with autonomic managers, non common actions should be introduced only when there is no natural way of mapping the desired functionality into one of the common actions defined in the `Effector` porttype. The current set of common actions defined in the `Effector` porttype are shown in Figure 7.

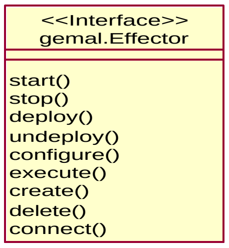

Figure 7: The `Effector` porttype class diagram.

2.2.1 The `EffectorProvider` class

The `EffectorProvider` is a Java abstract class which developers of managed elements extend to tailor the implementation of each common action to the relevant operation specific to the managed element. In addition to basic exception throwing of non supported actions, it provides utility methods to simplify the use of parameters in generic common actions.
In OGSi it is common to pass to each operation a single parameter: an XML document. This allows the definition of generic operation which can be customized at the implementation by passing as parameters different structures all represented as an XML tree. Although this approach is convenient for interface design, it is inconvenient to work with in the implementation where language specific data structures are easier to use.
In GeMaL we gave up some the flexibility of XML extensibility, and opted for a more programmer-friendly approach for generic parameter handling. Parameters to each common generic action are defined at the interface level as XML extensibility; but at the underlying java implementation, all common actions use a Properties list. Given that the name of the parameter to each action is specific to the action implementation and to the managed element, there is a need to register the keys of the Properties list. For this function a utility method `register` is provided by the `EffectorProvider`. This method should be called at the initialization stage of the concrete classes extending this class.

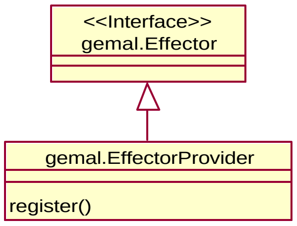

Figure 8: The `EffectorProvider` class diagram.

2.2.2 The `SinkEffectorProvider` class

In addition to the common actions, there are cases where we need to extend the `NotificationSink` porttype. This is needed to enable the flow of notifications from managed elements to autonomic managers (which also implement the `Effector` porttype) or to intermediaries (see section 3.1).

For those managed elements that need the `NotificationSink` functionality, GeMaL provides the `SinkEffectorProvider` class (see Figure 9). It extends the `EffectorProvider` to provide a default implementation of the `connect` operation: a call to this operation with a grid service handle as a parameter will cause the `SinkEffectorProvider` instance to subscribe to notifications from the corresponding grid service (assuming it is a notification source). . To enable selection of which events are to be sent to a `SinkEffectorProvider`, the `SelectionRule` service data element is used: if it is set to a valid XPath expression, then this expression is used on as a parameter on all subscriptions.

It also provides a default implementation of the `deliverNotification` operation: when it receives a notification it updates the `LastEvents` service data element

Naturally, a managed element for which this default behavior is not appropriate should extend the `SinkEffectorProvider` and override either operation (or both)

See section 3.1 for an example usage of this utility class.

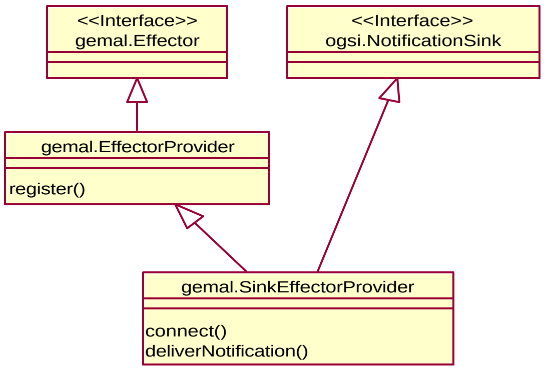

Figure 9: The `SinkEffectorProvider` class diagram.

2.3 Using GeMaL: Creating a Managed Element

Although autonomic systems can be built from scratch, we believe that the fist step toward autonomic computing will be making managed elements out of existing systems, i.e., we need to wrap systems/components with a well defined manageability interface. Using GeMaL this is a three steps process: 1) create a sensor for your system either by providing translation rules for the Generic Log Adapter; or by implementing your own sensor provider (which should implement the `SensorPortType` interface and extend the `NotificationSourceProvider`) and tailoring it to your application needs; 2) implement the relevant operations in your effector provider (which should extending the default GeMaL `EffectorProvider`); and finally, 3) creating a grid service deployment descriptor that puts together these operation providers. In this section we will go into what we did in each of these steps to create the Apace Managed Element.

2.4 The Apache Sensor

As mentioned in section 2.1.2, the preferred approach to create a sensor is to use the Generic Log Adapter. For Apache this means creating two set of parsing rules and configuration parameters: one for the `access_log` file and one for the `error_log` file. These rules are created using the graphical rule builder and configuration tool included with the Generic Log Adapter. The combination of rules and configuration parameters is a *GLA context*.; at runtime autonomic managers can select which context(s) to load by invoking the `collectEvents(contextName)` operation of the `GenericSensorProvider`. It is important to note that to enable the updating of the `LastEvents` service data elements all GLA contexts should include the definition of an additional GLA sink – the `gemal.GlaSink` as follow:

```
<com.ibm.acad.outputter:Sink
    class-
Name="com.ibm.ogsa.gemal.GlaSink”/>
    <com.ibm.acad.outputter:Config>
        agentName=CBEMsgAgent
    </com.ibm.acad.outputter:Config>
</com.ibm.acad.outputter:Sink>
```

Alternatively, we could write customized log adapters for our managed element. As in the previous approach, for Apache we need to deal with two log file formats hence we'll have two adapter classes: the `ApacheAccessAdapter` and the `ApacheErrorAdapter`. We also need to write our own operation provider: the `ApacheSensorProvider` (see Figure 10), which serves as the control point for the adapter classes. In this approach it is up to developer to manage the

`LastEvents` service data element. Given that autonomic managers expect a well defined behavior for this SDE (show a window with the *n* last events), the best approach is to reuse the code that already deals with this SDE updates, i.e., even in customized adapters it is recommended to use the `gemal.GlaSink.`

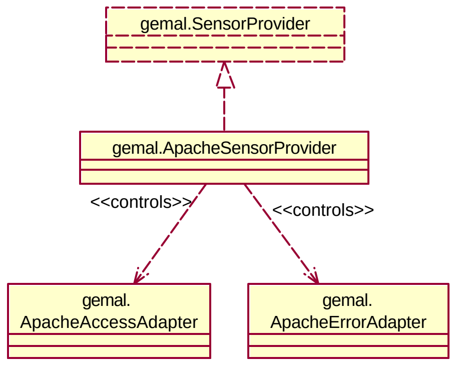

Figure 10: The `ApacheSensorProvider` class diagram.

2.5 The Apache Effector

We add the effector functionality to our managed element by extending the `EffectorProvider` and implementing the relevant operations: `start`, `stop`, `configure` and `connect`. While the implementation of the first three is pretty obvious, our implementation of the `connect` operation requires some additional discussion.

Going back to the multi-tier application show in Figure 2 and Figure 3 we can think of the `connect` operation as a method to add to the configuration of servers at tier *n*, a new server a tier *n+1*. In another words, we add an application servers to the apache configuration by invoking the `connect` operation on all apache managed elements. In our example we use Tomcat as the application server; adding a tomcat server to an existing apache serve involves adding a few lines to the apache configuration file. These lines include the address and port of the tomcat server and the prefix used by apache to identify the requests that need to be forwarded.

Hence the `connect` have three parameters which need to be registered when the `ApacheEffectorProviders` gets instantiated (see section 4). This is achieved by placing the following code in a static initializer:

```
String [] keys = [“host”,“port”,“prefix”];
register(“connect”, keys);
```

2.6 The Deployment Descriptor

In GT3, the set of classes that form a grid service instance are set in the deployment descriptor file. A grid service is normally composed of a base class which must be a derivative of the `GridServiceImpl` class included in GT3, and a list of operation providers.

In the deployment descriptor there are descriptions for all the persistent services a container has. Some of these services are factories that are responsible of creating grid services. A managed element is normally declared by describing the factory responsible for its creation as follow:

- Instance-Name – simple string to describe the Element
- Instance-schemaPath – a schema that describes the element's methods and service data
- Instance-baseClassName – the `GridServiceImpl` derivative.
- Instance-operationProviders - the effector and sensor concrete classes specific for this manages element

In the case of the apache managed element using the `GenericSensorProvider`, this leads to the following declarations in the deployment descriptor file (full package names were removed for clarity):

```
.
.
.
<parameter name="instance-baseClassName"
  value="…ogsi.GridServiceImpl" />
<parameter name="instance-className"
  value="…gemal.ManagedElementPortType" />
<parameter
  name="instanceoperationProviders"
  value=
    "…gemal.apache.ApacheEffectorProvider
     …gemal.GenericSensorProvider" />
.
.
.
```

3 Basic Modules for AC

3.1 Monitoring

In a distributed system a system level autonomic manager may need to collect data from several sub-components. Although this can be done by directly subscribing to, or querying, each sub-component, there are cases where is more convenient to aggregate the interesting data coming from all sub-components in a single repository, and then using this repository as the source of the data.

For this purpose we borrow the concept of a *Basket Service* from the Reporting Grid Service (ReGS) [2]. A basket is a configurable repository of CBE's that serves as an intermediary between managed elements and autonomic managers. Each basket has a filtering rule that determines what data goes into the basket; so we can create different basket to collect different pieces of information.

For a basic basket that serves only as an aggregator of data coming from different source we can use an instance of the `SinkEffectorProvider` directly since it already provides a filtering mechanism (through the `SelectionRule` SDE) and updates the `LastEvents` SDE on each notification. For a more sophisticated basket such as one that deals with persistency users need to extend the `SinkEffectorProvider` class. GeMaL includes a basket based on Apache Xindice XML database.

3.2 Execution

The GeMaL `EffectorProvider` focuses on exposing to autonomic managers the basic control and configuration functionality of the managed element. However, invocation of actions on remote managed elements requires from the autonomic manager to deal with several complex issues unrelated to its core problem analysis logic. To remove this complexity from the main code of autonomic managers, GeMaL provides separate functional module – the *GeMaL Coordinator*.

To better understand the kinds of issues the Coordinator needs to deal with, consider the following scenario: a number of web servers are grouped for achieving high reliability and load balancing. Each of these servers is connected to the same data base server.

It seems reasonable to have one autonomic manager for the database maintenance and control, and another one to perform cluster management and monitoring. Suppose that the cluster autonomic manager concludes that it should reconfigure all cluster components, including the database server, in order to adjust the system to the instant overload conditions. Such reconfiguration might come exactly during the periodic system maintenance of that database server, managed by another autonomic manager. Obviously collisions and conflicts are unavoidable, hence a mechanism, most likely based on a policy engine, to deal with conflicts and priorities is needed.

The detection of the data base failure will eventually lead to the autonomic manager fetching a batch of commands to be applied to a group of managed elements. Clearly, is not scalable for the autonomic manager to issue the commands one by one to each managed element. Instead, execution is delegated to the Coordinator, which takes care of the reliable execution of the commands, similar to the functionality provided by the Fault Tolerant Shell [4]. In addition, the Coordinator provides a single entry point to access multiple managed elements simultaneously, allowing full off-loading of the script execution overhead from the autonomic manager.

Finally, remote call invocation in distributed system is susceptible to network and software failures, which are related to the distributed infrastructure itself, and not to business logic. For instance, suppose the remote call to `start` function of Apache Managed Element (see 2.5) fails. It can fail due to the network failure during the call, or as a result of service container being too busy and dropping new connections. In both cases, the call was not executed by the remote managed element, and thus should probably be automatically retried later. However, if the call fails due to the managed element already being started by the previous calls, such call should not be retried, and the result should be returned to the caller.

At the time of writing, the GeMaL Coordinator is very simplistic: it does not provide any kind of invocation quality of service, or policy based coordination. It does, however, simplifies the work with the managed elements, hiding the complexity of dealing with the grid services. It implements FIFO scheduling, receiving a list of actions from an autonomic manager, and executing them sequentially in the order received.

As all components in our system, the Coordinator is a managed element. In the `CoordinatorEffectorProvider` all operations take as a parameter the name to a managed element and delegate the operations to it.

The `connect` operation is used to connect the Coordinator to a given service container and to initiate the discovery of the existing managed elements in it.

The `execute` call exposes the simple sequential script execution functionality, outlined in the previous subsection. For example the following execution script, connects to the given container, creates one managed element and starts it:

```
Execute(
       Connect(containerIP),
       Create(containerIP,
              MEname,createParams),
       Start(containerIP,MEname)
)
```

4 Putting it all together

As a proof of concept, we wrapped the with the generic manageability interface a set of varied components that allowed us to test the entire MAPE-K loop in a self healing scenario. Our demo scenario (see **Figure 11**) consists of several Apache servers connected to several Tomcat servers; we also have a Xindice based basket for collection of log data and the GeMaL Coordinator as the execution infrastructure. For our analysis and planning we use the Tivoli Autonomic Management Engine (TAME) [4].
In this environment we introduced configuration error which TAME at first was able to catch, but there wasn't enough information in the log data collected to clearly identify it. Consequently, TAME requested more data and then it was able to identify the source of the problem and issue all the corrective commands.
The actual problem determination logic in this scenario is not that complex but the main purpose of this demo was to show how GeMaL can be used to glue together autonomic systems, and to exercise the functionality of the basic modules provided with the library.

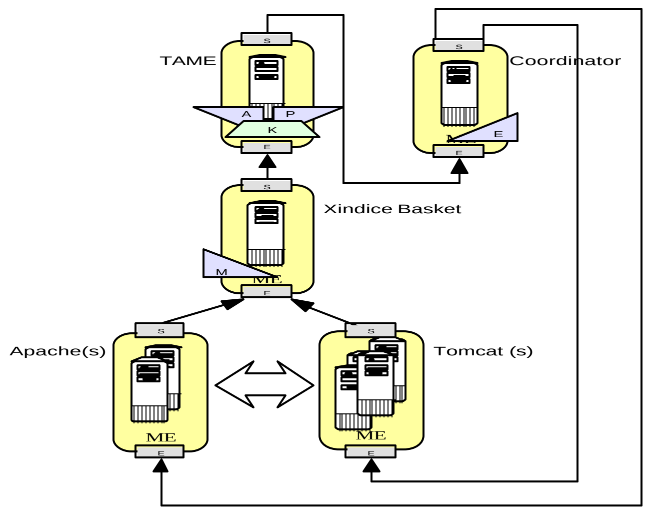

Figure 11: The GeMaL proof of concept scenario.

5 Conclusions and Future Work

The use of generic common actions has proven a valid concept that simplifies the development of both managed elements and autonomic managers. Adding new components to a system such as the one described in the previous section is very simple and can be done with very little programming.
The next step in the evolution of the system will be to enhance the coordination component to deal with all the complex issues such as reliable invocation and policy-based action coordination and synchronization. We also need to do some work in enhancing the functionality of our monitoring infrastructure.

6 References

[1] *An architectural blueprint for autonomic computing*, IBM, April 2003, http://www-3.ibm.com/autonomic/pdfs/ACwpFinal.pdf

[2] *Apache Xindice*, The Apache Software Foundation, http://xml.apache.org/xindice/

[3] Aridor, Horn, Lorenz, Rochwerger and Salem, *The Reporting Grid Services (ReGS)*, GGF, http://www.gridforum.org/Meetings/ggf7/drafts/draft-ggf-ogsa-regs-01.3.1.pdf

[4] Chase, *An Autonomic Computing Roadmap*, IBM, December 2003, http://www-106.ibm.com/developerworks/library/ac-roadmap/

[5] Foster, Gannon and Kishimoto, *The Open Grid Services Architecture*, Global Grid Forum, October 2003, https://forge.gridforum.org/projects/ogsa-wg/document/draft-ggf-ogsa-spec/en/13

[6] Grabarnik, Ma, Salahshour and Subramenia, *The Generic Adapter Logging Toolkit*, submitted to ICAC-04, http://www.alphaworks.ibm.com/tech/glaac

[7] *Java Programmer's Guide Core Framework*, Globus Alliance, September 2003, http://www-unix.globus.org/toolkit/3.0/ogsa/docs/java_programmers_guide.html

[8] Ogle, et al., *Canonical Situation Data Format: The Common Base Event*, version 2.10, IBM, October 2003, http://xml.coverpages.org/CommonBaseEventSituationDataV210.pdf

[9] *Red Hat Linux 9: Red Hat Linux Reference Guide*, http://www.redhat.com/docs/manuals/linux/RHL-9-Manual/ref-guide/s1-boot-init-shutdown-sysv.html

[10] Sandholm and Gawor, *Globus Toolkit 3 Core – A Grid Service Container Framework*, Globus Alliance, July 2003, http://www-unix.globus.org/toolkit/3.0/ogsa/docs/gt3_core.pdf

[11] Thain and Livni, The Ethernet Approach to Grid Computing. In Proceedings of HPDC, 2003

[12] Tuecke, et.al., *Open Grid Services Infrastructure (OGSI)*, Version 1.0, Global Grid Forum, June 2003, https://forge.gridforum.org/projects/ogsi-wg/document/Final_OGSI_Specification_V1.0/en/1