A Pattern Language for Adaptive Distributed Systems

Francisco José da Silva e Silva[1], Fabio Kon[2], Joseph Yoder[3], Ralph Johnson[3]

[1]Department of Informatics - Federal University of Maranhão
[2]Department of Computer Science - University of São Paulo
[3]Department of Computer Science - University of Illinois at Urbana-Champaign

fssilva@deinf.ufma.br, kon@ime.usp.br,
joe@joeyoder.com, johnson@cs.uiuc.edu

Introduction

Modern computing environments are characterized by a high level of dynamism. Two major kinds of dynamic changes occur frequently. The first refers to structural changes such as hardware and software upgrades, protocol and API updates, and operating system evolution. The second refers to dynamic changes in the availability of memory, CPU, network bandwidth and, in mobile systems, connectivity and location. Drastic changes may occur in a few seconds, impacting the performance of user applications profoundly. Among existing production software systems few offer support for managing, adapting, and reacting to these changes; most of the times, all the work is left to users and system administrators who must take care of them manually.

Fortunately, this scenario is gradually changing as researchers in academia and industry investigate elegant and robust ways to build self-adaptive systems for the dynamic, distributed environments of the future. In this paper we present a pattern language that captures some of the most relevant problems and solutions faced by developers who accept the challenge of building automatically configurable and adaptive distributed systems.

Dynamic Reconfiguration

Services must grow to meet increasing usage, new requirements, and new applications. However, flexibility usually conflicts with availability. In conventional systems, the service provider must often shut down, reconfigure, and restart the service to update or reconfigure it. In many cases, it is unacceptable to disrupt the services for any period of time. Disruption may result in business loss, as in the case of electronic commerce, or it may put lives in danger, as in the case of mission critical systems delivering disaster information, for example. Research in dynamic reconfiguration seeks solutions to this problem.

By breaking a complex system into smaller components and by allowing the dynamic replacement and reconfiguration of individual components with minimal disruption of system execution, it is possible to combine high degrees of flexibility and availability.

Self-Adaptation

Highly heterogeneous platforms and varying resource availability motivates the need for self-adapting software. Applications can improve their performance by using different

algorithms in different situations and switching from one algorithm to another according to environmental conditions. Significant variations in resource availability should trigger architectural reconfigurations, component replacements, and changes in the components' internal parameters.

Consider, for example, the network connectivity of a mobile computer as its user commutes from work to home. As the user switches from a wired connection at the office, to a wireless WAN using a cellular phone, and finally, to a modem connection at home, the available bandwidth changes by several orders of magnitude. The movement is also accompanied by changes in latency, error rates, connectivity, protocols, and cost.

Ideally, we would like to have a system capable of maintaining an explicit representation of the dependencies among the network drivers, transport protocols, communication services, and the application components that use them. Only then, would it be possible to inform the interested parties when significant changes occur. Upon receiving the change notifications, applications and services would be able to select different mechanisms, replace components, and modify their internal configuration to adapt to the changes, optimizing performance.

Designing Self-Adaptive Systems

The design of a self-adaptive system must answer three key questions:

1. **When to adapt?** How can the system detect that it is time to adapt (change its behavior) so that its performance will improve or that changes in the environment will not harm system correct functioning.
2. **What do adapt?** Which parts, elements, components of the system (e.g., mechanisms, algorithms and protocols) are subject to being adapted or replaced?
3. **How to adapt?** What are the mechanisms that allow for change in behavior? Given a certain system and environmental state, which adaptations would be more beneficial?

Only by addressing the three key questions above, a software framework can provide a comprehensive solution to the problem of building effective self-adaptive systems. In the remaining of this paper we present a pattern language that addresses the most important aspects of dynamic reconfiguration and adaptation in distributed systems.

Note, however, that there are other important non-functional aspects that are orthogonal to the patterns in this language. Aspects such as Security, Fault-Tolerance, and Real-Time can be essential factors for consideration depending upon the different environments the system will be deployed. In these cases, the reader should refer to patterns specific to these domains.

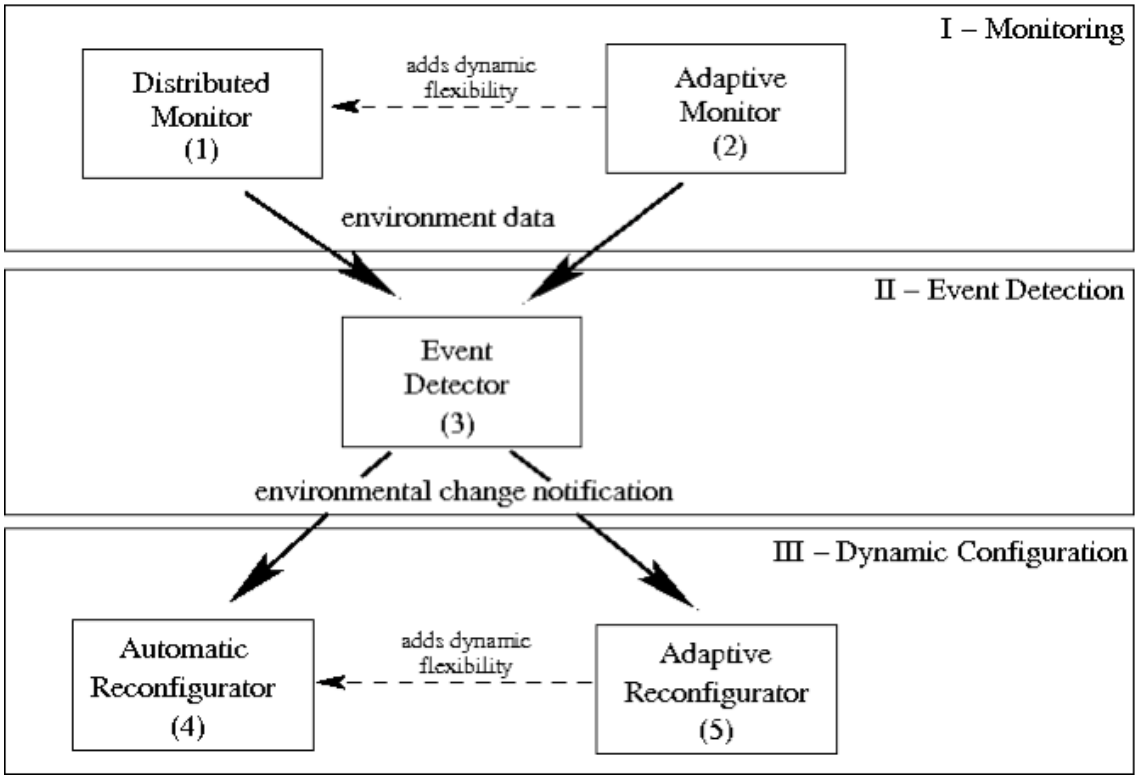

Figure 1: Pattern Language Structure

Figure 1 illustrates the pattern language structure, which is composed of three parts: monitoring, event detection, and dynamic configuration. Part I helps answer the "**When**" question. These patterns are related to monitoring distributed environments. The **DISTRIBUTED MONITOR (1)** and the **ADAPTIVE MONITOR (2)** patterns describe monitoring solutions for applications that must obtain a global view of the state of distributed resources to decide when adaptations should be performed. The **DISTRIBUTED MONITOR (1)** provides a simpler solution while the **ADAPTIVE MONITOR (2)** describes an extension that supports dynamic reconfiguration of the monitor. Both monitors provide monitoring data to event detection mechanisms. Part II presents the **EVENT DETECTOR (3)**, which helps detect when an adaptation should be performed and decides "**What**" adaptations should be performed. When the need for an adaptation is detected (with the information provided by the monitors), the **EVENT DETECTOR (3)** notifies the mechanisms responsible for the dynamic reconfiguration of the system.

Part III shows "**How**" adaptation can be performed with the **AUTOMATIC RECONFIGURATOR (4)** and the **ADAPTIVE RECONFIGURATOR (5)** patterns. Again, the former pattern describes a simpler solution while the latter describes an extension that supports dynamic reconfiguration of the reconfiguration process, leading to a "reconfigurable reconfigurator".

These patterns work together to solve the problem of describing what resources we are concerned about, when to adapt and how to adapt. We now present the five patterns of the pattern language. The patterns are presented in the same order in which information in the system flows, i.e., the monitors collect information, passing it to the detector, which, in its turn, notifies the reconfigurators.

1. DISTRIBUTED MONITOR

Motivation:

In order to improve its performance by means of dynamic configuration, a system must be aware of the dynamic state of the environment in which it is executing. In a distributed system, the environment is spread throughout a collection of, possibly heterogeneous, machines linked by, possibly heterogeneous, network links. Monitoring resource availability can help detect when the system reaches a state in which dynamic reconfiguration would improve application performance or avoid its breakage

Problem:

How to monitor the resources of a distributed system efficiently?

Forces:

- The state of a distributed system is composed of many variables distributed across many machines in various locations. The communication delays and relative speeds of computations of asynchronous distributed systems make it difficult to detect a global state in which dynamic adaptation would be desirable.
- Trying to detect opportunities for dynamic adaptation by looking at isolated machines is easier but this approach cannot provide optimal solutions; it is very likely that relevant global information will be missing.
- Having a single centralized node be aware of the state of the entire system might be infeasible since this compromises scalability (the central node becomes a bottleneck as the system grows) and fault-tolerance (the central node becomes a single point of failure). One can approximate a centralized view of the global state by sending messages from all the nodes to the central node at a high rate. But this may impose an extremely high communication cost and make the central node a single point of failure.
- On the other hand, replicating the central node to avoid a single point of failure increases system complexity and network usage.
- Adopting a lazy protocol in which the state of individual nodes is sent to the central server at a slow rate could solve the network congestion problem but would probably make the data in the central node stale and therefore of little use.

Solution:

Organize the distributed system as a hierarchy of clusters, as illustrated in Figure 2, so that each cluster includes the machines in a local area network, typically containing from a few to approximately one hundred machines (this number can vary depending upon requirements and the environment). Provide a single (possibly replicated) Monitoring Server for each cluster. Have the cluster nodes send periodic information about their local state to the Monitoring Server, as illustrated in Figure 3. These messages may be sent by multicast to all replicated copies of the Monitoring Server (e.g., using the IP-Multicast protocol) so that the network load is not increased by an increase in the number of replicas.

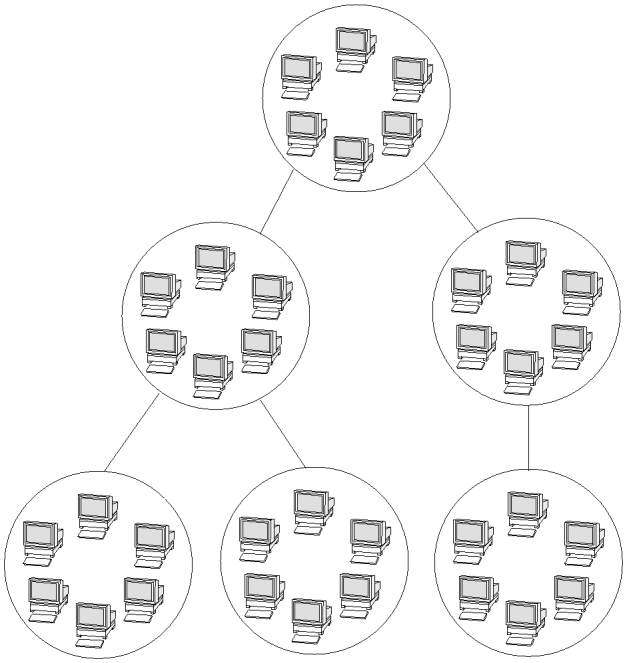

Figure 2 – Hierarchy of computer clusters

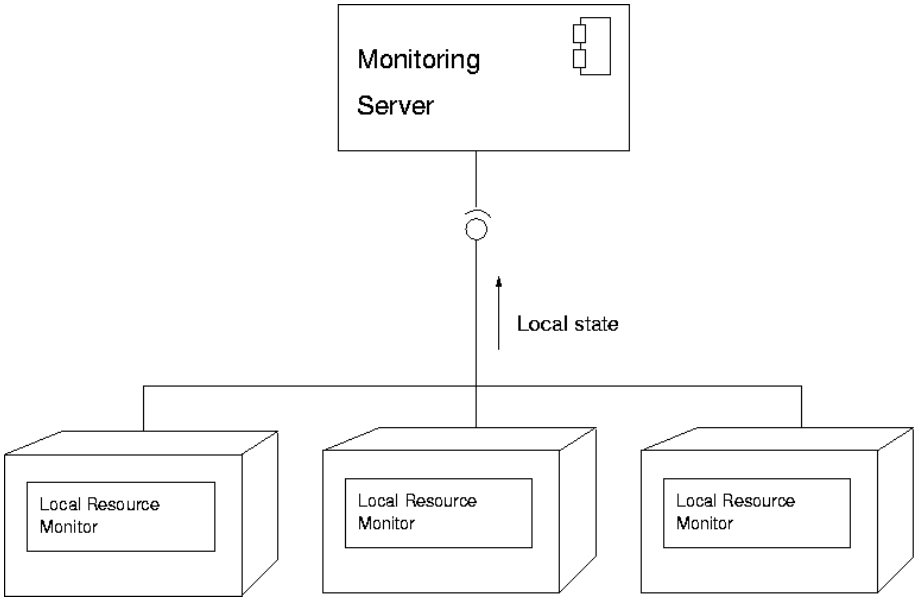

Figure 3 – Distributed Monitoring within a single cluster of machines

To avoid unnecessary messages in the network, the frequency with which the messages are sent to the Monitoring Server cannot be high. Thus, have each node monitor its resources locally at a higher rate (e.g., once per minute) and only send update messages to the Monitoring Server when a significant change in the local state occurs. If no changes occur during a long period (e.g., 5 minutes) then send an update message to the Monitoring Server as a keep-alive. If it is not important whether the nodes are alive or not, then this keep-alive message is not necessary.

The Monitoring Servers of different clusters may be organized in a hierarchy so that consolidated information about cluster state can be exchanged across clusters in a lazy fashion. The farther a Monitoring Server is from a certain machine, the less accurate this monitoring information will be for this machine since changes might have happened to the machine by the time the Server processes the information.

Example:

A distributed system is composed of several distributed resources, such as CPUs, memory, disks, and network links. For each resource, one would like to know the current usage level. Resource usage can be expressed by several properties. For a network link, for instance, properties could be available bandwidth, current latency, number of collisions, etc. In such a system, it would be desirable to have a load balancing mechanism that would migrate tasks from one machine to another depending on resource availability on the distributed system.

Consequences:

+ Network usage is limited (can be fine tuned through the periodicities).
+ One can have a good approximation of the system global state.
_ Implementation in the hierarchical case can be complex, compared to a single centralized server.
_ To implement the monitoring service component that collects the state of local resources in each node one must define, in advance, which machine resources will be monitored.

Resulting Context:

By applying this pattern, a collection of machines, possibly organized in a hierarchy of machine clusters, can be monitored with low network and processor overheads. Thus, it is possible to get an approximate view of the global state of the resources in the distributed system. This pattern requires that the kinds of resources to be monitored be defined *a priori*; if there is a need for adding new types of resources to be monitored at runtime, then one should use the **ADAPTIVE MONITOR (2)** instead. This pattern describes how to collect information about "**When**" to adapt, which will be used by the **EVENT DETECTOR (3)** for triggering the adaptations.

Related Patterns:

- The Publisher-subscriber pattern [Buschmann:1996] describes a mechanism for objects in a distributed system to declare interest in receiving information about a certain topic and for publishing information to be sent to the interested objects.

Known Uses:

- Grid computing systems instantiate this pattern to monitor the geographically distributed machines of the Grid. The Globus toolkit [Foster:1997], for example, uses the LDAP protocol for communicating information about resource availability and status of Grid nodes; a federation of LDAP servers plays the roles of the monitoring servers of this pattern. The InteGrade Grid middleware [Goldchleger:2003] also instantiates the pattern but uses CORBA for communication and a new service, called Global Resource Manager, as the monitoring server.
- The 2K operating system [Kon:2000, Kon:2005] instantiates this pattern to maintain an approximate view of the state of machines in the distributed system and uses this view as a hint for remote execution of user applications.
- The Framework for Adaptive Distributed Systems [Silva:2003] developed by Silva in his PhD work provides a generic object-oriented framework for instantiating this pattern based on CORBA distributed objects. Communication is performed with the CORBA event service, which is an instantiation of the Publisher-Subscriber pattern [Buschmann:1996].

Variant:

For some adaptive distributed applications there is no need for a centralized view of the state of distributed resources (e.g., a video client retrieving a movie from a video server). Instead, the application is only concerned with the state of resources located in the path between its components. In such a case, there is no need for a Monitoring Server. Each software component responsible for monitoring a resource should implement an interface through which the state of the resource can be queried. It should also implement a notification service through which applications can register interest in being notified about changes on the state of the monitored resource.

Implementation:

To implement this pattern, a system developer must implement and deploy Local Resource Monitors for each of the resources to be monitored. When implementing monitors for resources such as CPU and memory, almost always it is necessary to deal with the specificities of each operating system as there are no widespred standards for getting this kind of information. In Linux systems, for example, it is common to use the `/proc` pseudo-filesystem to get information about resource usage such as CPU and memory. In Windows systems, it is common to rely on Win32 API functions such as `GlobalMemoryStatus` to obtain memory information and `RegQueryValueEx` to get CPU consumption information from the Windows registry.

Local Resource Monitors must send data updates to the Monitoring Server periodically. This one-way communication can be implemented in various ways: from rudimentary sockets (using UDP, TCP or IP-Multicast channels) to higher level middleware mechanisms such as Java RMI, CORBA IIOP, or SOAP.

2. ADAPTIVE MONITOR

Motivation:

In very dynamic systems, we may not know *a priori* which objects and resources should be monitored. As new applications are installed in a system, we may need to monitor information that was previously irrelevant or not available.

Problem:

How to monitor system objects and resources that are not known at the design phase of the monitoring system?

Forces:

- Limiting the types of objects and resources that are monitored makes it easier to develop adaptive strategies because there is less information to be managed and analyzed. However, it is hard to predict ahead of time which objects and resources are available or needed as a system or demands on the system evolves.
- Systems, services, and resources change often as the requirements evolve. The monitoring system must cope with such changes by starting to monitor new things and stopping the monitoring of things that are no longer relevant.
- A distributed system is composed of several distributed resources, such as workstations, network links, and application servers. The usage of a resource can be expressed by different properties that adaptive applications might be interested in monitoring. For a network link, for instance, properties could be available bandwidth, current latency, and number of collisions. Therefore, a flexible way to describe distributed resources and their monitoring properties is needed.

Solution:

Define an Object Monitor responsible for gathering the state information of a single resource property and allow dynamic loading and unloading of these monitors in the various nodes of the distributed system. Each resource property could be defined with three attributes: resource name, property name and value type (e.g., <"Network Link", "Available Bandwidth", "Mbps"> <"CPU", "CPU load", "percentage">).

Define a standard Object Monitor interface regardless of the property to be monitored and define a standard protocol and message format to be supported by the Monitoring Server. Each Object Monitor registers itself with the Monitoring Server as part of its instantiation process. Object Monitors send a message to the Monitoring Server whenever there is a significant change on the state of the resource properties they monitor, as illustrated in Figure 4.

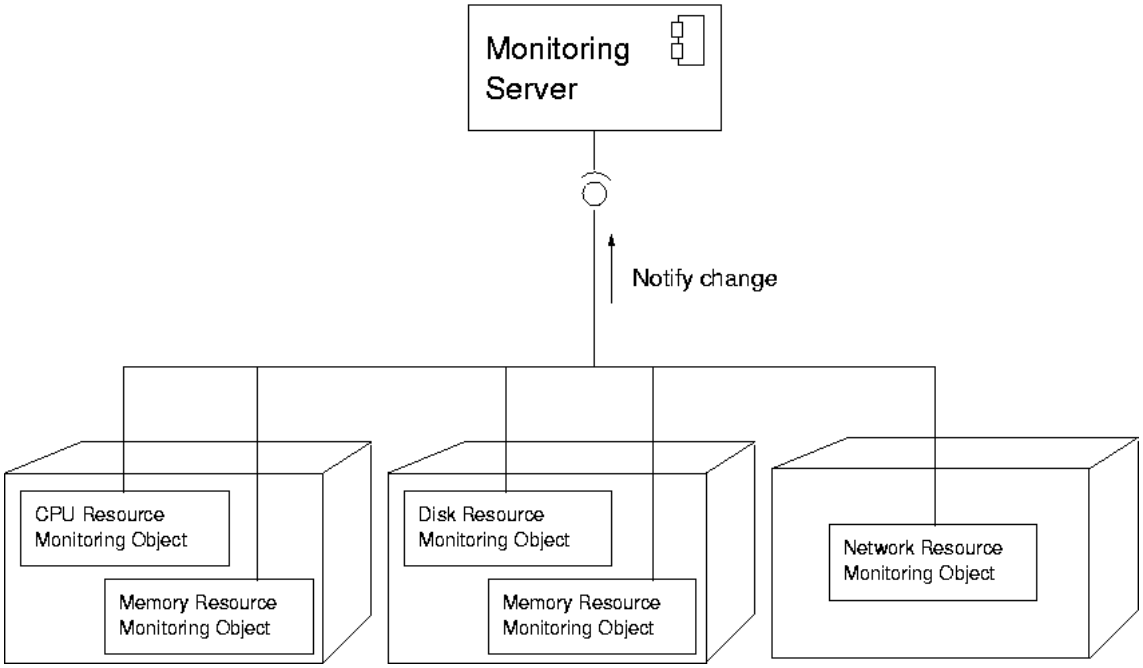

Figure 4 – Monitoring different kinds of resources

Example:

Consider the example described in the **DISTRIBUTED MONITOR (1)** pattern, where the CPU and memory of distributed machines are monitored to provide load balancing. When a machine becomes congested, with high CPU or memory usage, the load is balanced by migrating tasks to a machine with lower load. If new applications composed of several collaborative tasks must now be executed, a new resource property (available bandwidth) should also be monitored to avoid allocating collaborative tasks to different machines connected by low bandwidth links. Therefore, a new Object Monitor must be implemented (and dynamically loaded into the system) to monitor the available bandwidth.

As another example, consider a Web application for a bookstore. The load balancing among application servers can be based on CPU load. If, during the application execution, the administrator realizes that the network (and not the CPU) has become the bottleneck, an Object Monitor can be dynamically loaded into the application servers to redistribute the load based on the number of network connections rather than using just the CPU load as the scheduling parameter.

Consequences:

+ New or different resource properties can be monitored without affecting the code of the monitoring infrastructure
+ The monitoring service can start monitoring new or different resource properties without interrupting the service
– New resource properties should apply to the standard Object Monitor interface, which can limit the expressiveness of the resource property to be monitored

Implementation:

To implement such monitoring functionality, define a Resource Monitoring Object (RMO) that monitors a specific resource property. Each RMO monitors a single resource property that can correspond to physical resources such as memory, CPU, disk, and network links, but it can also monitor software parameters, such as the number of open threads in a server object.

Every resource property has a set of associated operation ranges, which are defined by the application developer. For example, one could use the following operation ranges for monitoring percentage of processor utilization: [0%, 10%), [10%, 25%), [25%, 50%), [50%, 75%), and [75%, 100%].

In all hosts containing resources that must be monitored, instantiate a Resource Monitoring Object for each resource property to be monitored. The RMO periodically verifies the current operation range of the resource property and only notifies registered components about changes on the operation range, limiting the number of monitoring messages in the distributed system. Figure 5 shows the RMO interface using CORBA IDL.

```
interface Rmo {
  MonitoredEntities::Parameter  parameter ();
  MonitoredEntities::Entity     me ();
  unsigned long    frequency ();
  unsigned long    current_range ();

  void  suspend ();
  void  resume ();
  void  change_frequency (in unsigned long new_frequency);
  oneway void  shutdown ();
};
```

Figure 5 - Resource Monitoring Object interface

The `parameter()` method returns a reference to the resource property being monitored while `me()` returns a reference to the monitored entity. `current_range()` returns the current operation range of the monitored property, allowing the developer to use, optionally, a pull approach in complement to the push mechanism described above. As an example, consider the operation ranges described above for monitoring percentage of processor utilization. The method `current_range()` would return 2 if a monitored processor usage is in the range [10%, 25%). The RMO interface also allows temporary suspension of the monitoring process (`suspend()`) as well as its resumption (`resume()`). `change_frequency()` alters the frequency used for verifying the operation range and `shutdown()` stops the monitoring process.

Figure 6 presents the class diagram for a Resource Monitoring Object responsible for monitoring the CPU usage on a host. The design is extensible, allowing the developer to construct easily new RMOs for monitoring other resource properties. To do so, the developer has to rewrite two classes, fully reusing the other five ones.

The `RMOImpl` class implements the Resource Monitoring Object interface illustrated in Figure 5. `CpuMonitor` and `RmoCpuImpl` are specific for the CPU usage property. The `CpuMonitor` class contains the code that actually verifies the

CPU usage with a given frequency encapsulated by the `Frequency` object. The user can change the frequency value through the `change_frequency()` method of the `RMOImpl` object. This method calls a `set()` method of the `Frequency` object. The user can also suspend or resume the monitoring by calling the `suspend()` and `resume()` methods of the `RMOImpl` object. These methods call the `SuspendMonitor` object that implements a monitor used by the `CpuMonitor` thread to verify if it must continue the CPU monitoring at the end of every monitoring interaction. A `CurrentRange` object encapsulates the value of the latest CPU usage calculated. The user can check the current CPU operation range by calling the `RMOImpl current_range()` method. The `Notifier` thread is responsible for sending a message to the Monitoring Server whenever there is a change on the CPU usage operation range.

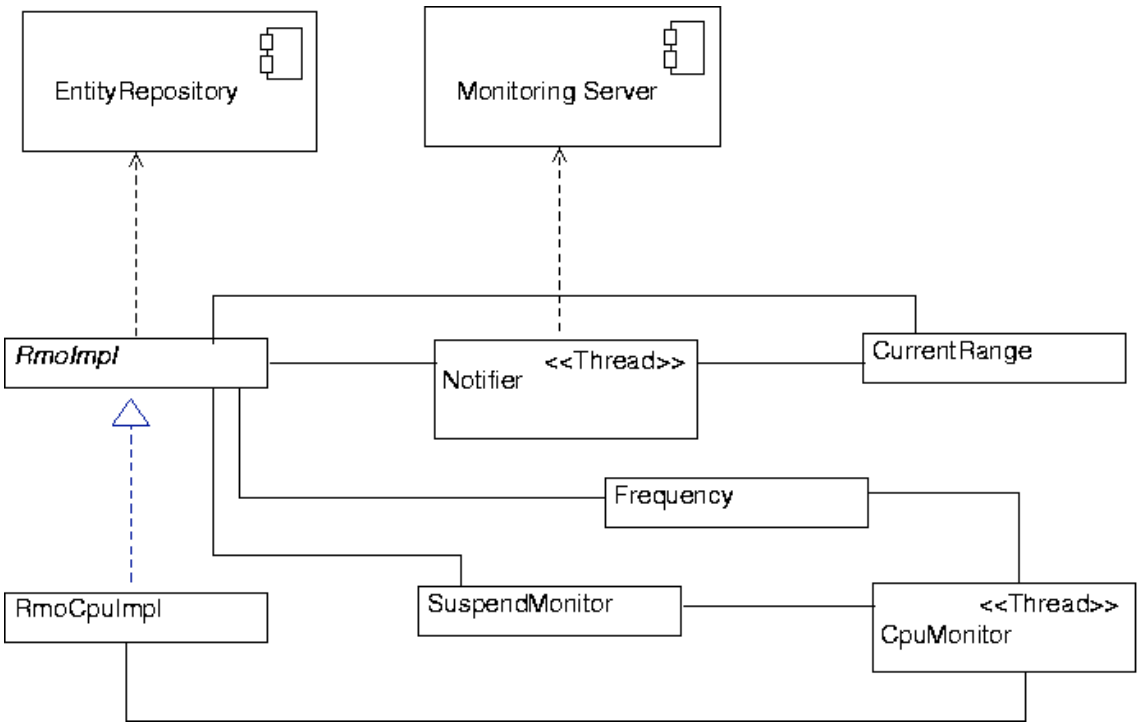

Figure 6 - Resource Monitoring Object monitoring CPU usage on a host

Resulting Context:

By applying this pattern, a collection of machines can be monitored with low overheads, enabling the construction of an approximate view of the global state of distributed resources. With the **ADAPTIVE MONITOR (2)**, the set of resources and the type of resources that are monitored can be reconfigured at runtime, enabling the monitoring of resources that were not anticipated at design time. This solution provides a large degree of flexibility. The data provided by the monitor will be processed by the **EVENT DETECTOR (3)**, which will trigger the adaptation actions.

Related Patterns:

- The Component Configurator pattern [Schmidt:2000] describes a mechanism for dynamically loading and configuring components into a running execution environment. This pattern can be used to load new monitoring objects dynamically.
- The TypeSquare pattern commonly used in Adaptive Object-Models (AOM) can be used for implementing the resource properties that can monitored [Yoder:2001; Yoder:2002]. The resource properties can be stored in a XML file that can be read at run-time in order to dynamically build the objects responsible for monitoring new defined resource properties without the necessity of recompiling and restarting the LocalResourceManager. AOM describes how to read the metadata file and dynamically build these objects using the Interpreter and Builder patterns [Gamma:1994].
- The Properties Pattern can be used for implementing different types of resource properties that are monitored [Foote:1998; Yoder:2001; Yoder:2002].

Known Uses:

- The Framework for Adaptive Distributed Systems [Silva:2003] allows specifying which resources will be monitored and how they will be monitored dynamically. This is achieved by dynamically loading new monitoring objects into the system runtime.
- The QuO Quality Objects Framework [Zinky:1997, Vanegas:1998, BBN:2002] provides a powerful CORBA-based framework for building quality of service aware, distributed applications. It instantiates this pattern using "system condition objects" as adaptable monitors.
- A mechanism for providing network environmental information in mobile wireless networks [Sudame:1997].
- An environment to support dynamic adaptation of distributed applications using the LuaORB system [Moura:2002].

3. EVENT DETECTOR

Motivation:

By analyzing the data provided by a distributed monitoring system, it is possible to identify relevant changes on resource availability that would impact application performance. By sending notifications to interested parties, it is possible to allow adaptive applications to reconfigure themselves to improve their performance in face of environmental changes.

Problem:

How to detect and notify applications about changes in the environment?

Forces:

- Tightly coupling the code responsible for detecting environmental changes with the application code adds unnecessary complexity, making the application code harder to implement, debug, and maintain. It also does not allow sharing the code with other environment-aware applications executing on the distributed system.
- On the other hand, each adaptive application can have specific needs concerning which environmental changes are relevant for dynamically adapting it.
- The notification of some environmental changes should be treated differently from others, leading to the need to apply different notification policies in different cases.

Solution:

An adaptive application must be notified of relevant changes in resource availability. These notifications can be implemented as asynchronous events. Expand the Monitoring Server interface (from the **DISTRIBUTED MONITOR (1)**) to allow the definition of event evaluators through conditional Boolean expressions. The Boolean expression indicates changes on resource property state. For instance, a "heavy use" event can be triggered when the percentage of the CPU usage on a host becomes greater than 80%.

Some applications need to correlate multiple events, such as the percentage of CPU usage and the amount of main memory available on a host. If this is the case, the definition of event evaluators must support composite events by allowing the Boolean expression to be composed of several resource properties.

Define an Event Channel to bind the event producer (Extended Monitoring Server) and event consumers (adaptive applications). The Event Channel implements the event delivery policy. You can organize your system with more than one Event Channel, each one responsible for notifying related events. For instance, a network channel may be associated with all network related events while a hardware channel may be associated with all events related to changes on hardware components. Figure 7 shows an abstract diagram of this architecture.

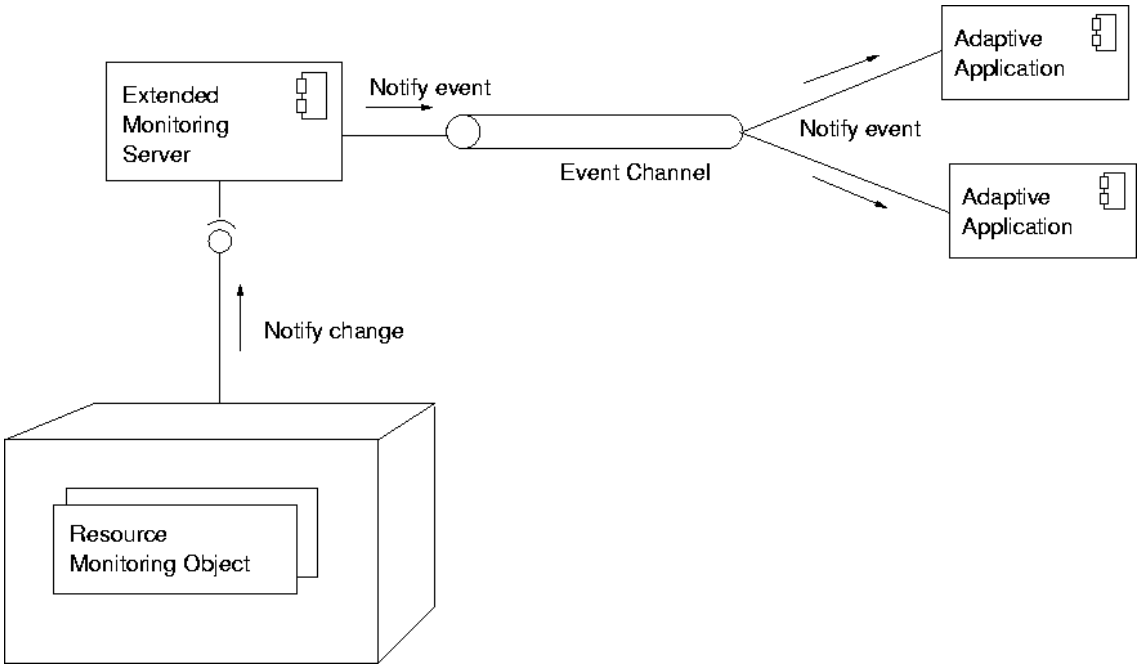

Figure 7 – Event Detector structure

Figure 8 illustrates the interactions between the pattern components. The adaptive application registers itself with the Event Channel, passing the list of environmental changes (events) in which it is interested. The resource monitoring objects (RMOs) continuously monitor the distributed system resources. Each RMO monitors a specific system parameter, notifying the Monitoring Server whenever a significant change is detected. The Monitoring Server evaluates all Boolean expressions containing the notified parameter and notifies the Event Channel whenever a Boolean expression is evaluated to true, meaning that an event has been detected. The Event Channel then notifies all adaptive applications that registered interest in the event that was triggered.

Example:

Consider a distributed system whose goal is to provide load balancing by migrating tasks from one machine to the other by looking at machines with congested CPUs and high memory usage. In each machine, instantiate two Object Monitors for monitoring the percentage of CPU load and the amount of memory available. Through the Monitoring Server interface, define a new Boolean expression that triggers an event every time a machine becomes congested, such as: `CPU_load > 80%` and `memory_available < 20MB`. Define an event channel used to notify the occurrence of events. The event channel abstraction is provided by some distributed object middleware services, such as the CORBA event service [CORBA:2002]. The channel uses a push approach, where the Monitoring Server registers itself as an event producer and the components of adaptive applications responsible for the dynamic reconfigurations register themselves as consumers.

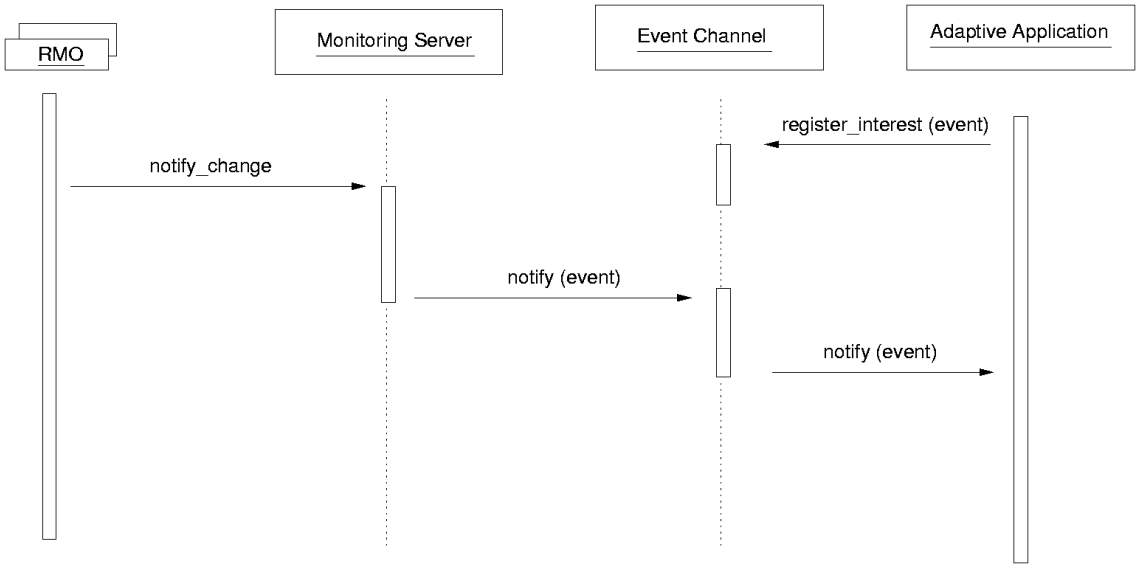

Figure 8 - Event detection and notification

Consequences:

+ Applications can share event evaluator definitions but also define new ones.
+ Allows flexible, event-specific delivery policies.
– Separating the code responsible for detecting environmental changes from application code adds complexity and can also lead to communication delays if they are deployed on separate machines.

Implementation:

Define a Monitoring Server responsible for collecting notifications of changes on the operation range of resource monitoring properties being monitored by Resource Monitoring Objects (from the **ADAPTIVE MONITOR (2)**). The Monitoring Server must allow the definition of events that are triggered based on a Boolean expression. Figure 9 shows a CORBA IDL interface for defining such resource events.

```
interface Event {

  string  eid();
  string  description();
};

interface ResourceEvent : Event {
  string                    expression ();
  unsigned long             duration_time ();
  MonitoredEntities::EntityType metype();
};
```

Figure 9 - IDL interface for an Event

`duration_time` specifies the amount of time that the Boolean expression must hold true to trigger an event notification. This avoids the notification of false events, based on temporary situations such as a short peak on CPU usage that occurs when a heavy application is started.

Since in a distributed system it is not guaranteed that messages are delivered in the same order that they were generated, each message from a RMO must include a timestamp. The Monitoring Server must maintain the last timestamp received from each RMO, discarding older messages without processing them.

Figure 10 shows the Monitoring Server interface. The `register()` method allows registering an event type and starts its detection. The interface allows the suspension and restart of the detection process though the `suspend()` and `resume()` methods, respectively. The `unregister()` method stops the detection of a given event type. An RMO notifies changes on the operation range of the resource being monitored through the `change_parameter()` method. To do so, it has to be previously registered through the `rmo_register()` method. If the execution of an RMO is suspended, resumed, or stopped the Monitoring Server must be informed through the `rmo_suspend()`, `rmo_resume()` and `rmo_unregister()` methods, respectively. The reason to do so, is that if an RMO is suspended or stopped, the Monitoring Server should not evaluate the event types whose Boolean expression contains the resource property that is no longer being monitored.

```
interface MonitoringServer{
      void register    (in ResourceEvent re);
      void unregister  (in string eid) raises(NoSuchEvent, EventNotBeeingEvaluated);
      void suspend     (in string eid) raises(NoSuchEvent, EventNotBeeingEvaluated);
      void resume      (in string eid) raises(NoSuchEvent, EventNotBeeingEvaluated);

      void change_parameter (in string meid, in string pid,
                             in unsigned long new_range);

      void rmo_register   (in string strRmo, in string meid, in string pid);
      void rmo_unregister (in string meid, in string pid);
      void rmo_suspend    (in string meid, in string pid);
      void rmo_resume     (in string meid, in string pid);
      range_list list_me_parameter_range(in string meid);
    };
```

Figure 10 - MonitoringServer interface

The MonitoringServer interface uses the following terminology: a monitored distributed resource is called a monitored entity. An object representing every monitored entity and all its monitored parameters must be previously created through an Entity Repository. The Entity Repository will create a globally unique identifier for monitored entities and parameters. In the `MonitoringServer` interface, `eid` stands for event identification, `meid` stands for monitored entity identification and `pid` for parameter identification.

Figure 11 shows the class diagram of the Monitoring Server implementation. The `MonitoringServerImpl` class acts as a mediator. It receives information about resource utilization from remote `RMO`s and maintains a local list of entities being monitored (instances of the `MonitoredEntity` class, which contains a description of the entity and its current value). `MonitoringServerImpl` also keeps a list of all active events whose descriptions are instances of `EventDescription`. An instance of the `Calculator` class performs the evaluation of the Boolean expressions defined in the event descriptions. Each time a Boolean expression is evaluated to true, a corresponding `SatisfiedEvent` is constructed. Once every second the `EventNotifier` checks for events whose expression holds true for the duration specified in the event definition. It then uses an Event Service (e.g., the CORBA one) to send notifications on event channels to which adaptive applications can listen to receive the notification of event occurrences.

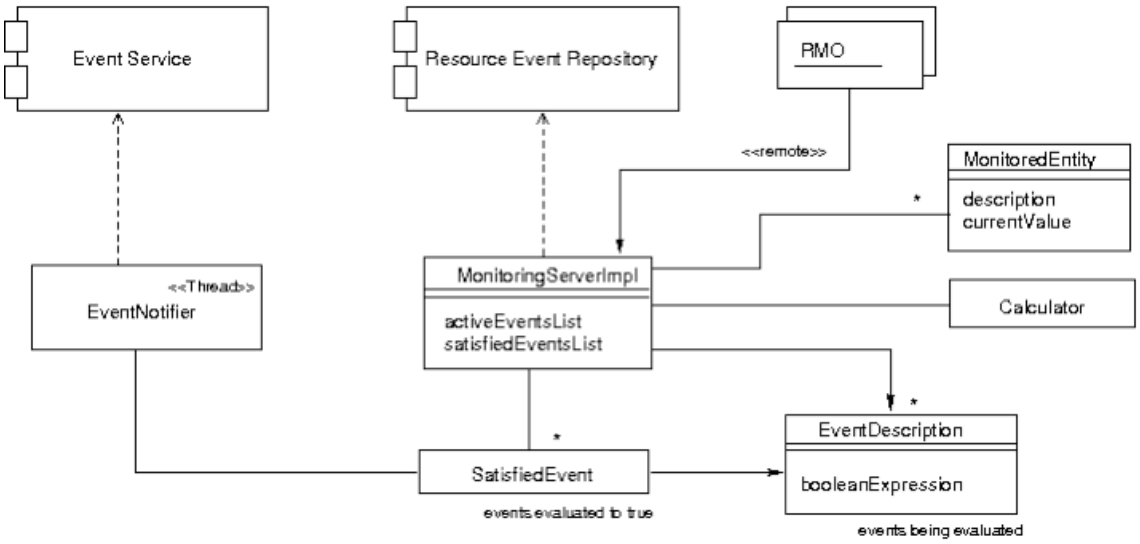

Figure 11 - Structure of the Monitoring Server implementation including event detection

Resulting Context:

By applying this pattern, it is possible to detect the occurrence of events based on the state of resources in a distributed system and to notify interested parties. In particular, the **AUTOMATIC RECONFIGURATOR (4)** and the **ADAPTIVE RECONFIGURATOR (5)** patterns rely on this event notification for carrying out the dynamic reconfiguration to adapt the applications.

Related Patterns:

- The instantiation of the Publisher-Subscriber pattern [Buschmann:1996] depends on a mechanism for matching subscriptions descriptions and event descriptions. This mechanism can be implemented by using the **EVENT DETECTOR (3)** described here.
- The Observer design pattern [Gamma:1994] describes an even simpler notification mechanism that can be used in some cases.
- The TypeSquare pattern described in Adaptive Object-Models (AOM) can be used for implementing the set of events that can change at runtime [Yoder:2001; Yoder:2002]. Event definitions can be stored in a XML file that can be read at run-time in order to dynamically build the objects responsible for detecting new defined events without the necessity of recompiling and restarting the MonitoringServer. AOM describes how to read the metadata file and dynamically build these objects using the Interpreter and Build patterns.
- The Interpreter design pattern [Gamma:1994] define a representation for a given language grammar along with an interpreter that uses the representation to interpret sentences in the language. It can be used for implementing the Boolean expression evaluator.

Known Uses:

- The Framework for Adaptive Distributed Systems [Silva:2003] includes an engine for event detection that instantiates this pattern faithfully.
- The QuO Quality Objects Framework [Zinky:1997] uses "delegates" and "contracts" to instantiate this pattern. Delegates act as proxies [Gamma:1994, Buschmann:1996] that intercept remote method calls; during interception, a delegate evaluates a contract to detect possible contract violations, which can be seen as a form of event detection.
- Moreto and Endler [Moreto:2001] describe a general purpose Event Processing Service (EPS), which can be used to detect primitive and composite events. Composite events are defined through an event expression based on primitive event types combined by a set of operators, similarly to the **EVENT DETECTOR (3)**.
- Welling and Badrinath [Welling:1997] describes an architecture for exporting environment awareness to mobile computing applications. In their architecture, a change in the environment is modeled as an asynchronous event that includes information related to the change. The architecture also allows alternate event delivery policies by isolating the event delivery functionality within a channel, as done in the **EVENT DETECTOR (3)**.

4. AUTOMATIC RECONFIGURATOR

Motivation:

A distributed system has many resources whose availability and load vary intensely. To cope with these variations, an adaptive application must be able to reconfigure itself as relevant changes on resource availability occur. Changes on resource availability could be notified through an event distribution mechanism. For each event, the set of adaptive actions that must be performed may vary.

Problem:

Given event notifications indicating changes on resource availability, how can a system apply dynamic reconfiguration actions automatically without the need for any human interference?

Forces:

- Coupling the application functional code with the code responsible for dynamic adaptation increases code complexity, making it harder to implement, debug, and maintain.
- The application developer should concentrate on the application core functionality, considering the adaptation issues as a separate aspect.
- Limiting which adaptation mechanisms can be applied (e.g., adjusting application parameters, switching between algorithms and relocation or replication of application components) restricts the solution applicability.

Solution:

Using a reflective model [Maes:1987], organize the application in two levels: a meta-level composed of objects responsible for receiving notifications of events describing environmental changes and for applying the reconfiguration actions and a base-level, that deals with regular application functionality.

For each event type indicating environmental changes that requires adaptation, describe the reconfiguration action(s) that your application must apply and code it into objects (called Handlers) that will be part of the meta-level. Each Handler object must implement a `run()` method, responsible for applying the reconfiguration actions when called. As illustrated in Figure 12, the Event Handler registers itself with the Event Channel (described in the **EVENT DETECTOR (3)**). Through the Event Channel, it receives notifications of environmental changes and reacts to them by applying the reconfiguration actions coded in its `run()` method.

Adaptive actions can be based on several mechanisms, such as:

a) Adjusting parameters of base-level objects (e.g., changing the presentation rate of video frames as network bandwidth varies);

b) Switching between algorithms used by base-level objects (e.g., changing a compression algorithm as CPU usage and network bandwidth varies);

c) Relocating or replicating application components to other network nodes.

In the application meta-level, instantiate an active object that registers itself as a consumer of the network event channel responsible for notifying environmental changes. Upon the receipt of an event, it calls the `run()` method of the corresponding Handler object.

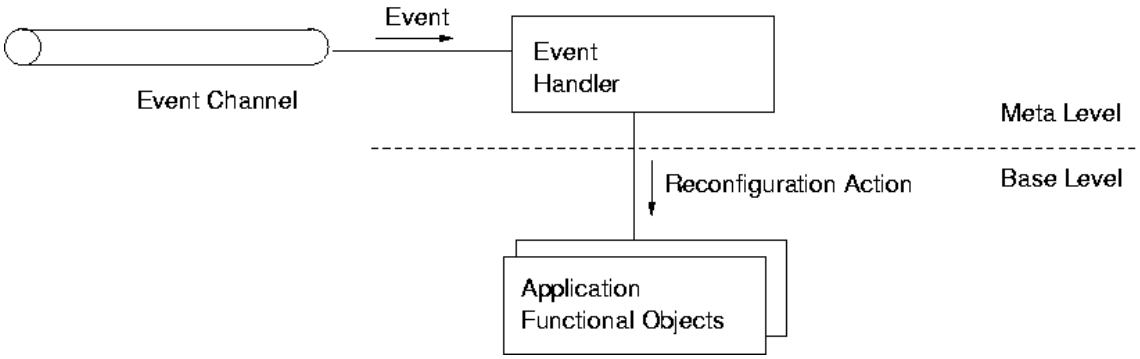

Figure 12 – Automatic Reconfigurator structure

Example:

Consider again the load balancing problem based on machine load. In the application meta-level, instantiate an object that registers itself as a consumer of the event channel responsible for notifying environmental changes. The adaptive application must react to a notification of an overloaded machine by migrating a subset of the tasks executing at the congested location to a machine with better CPU and memory availability (if one is available). The migration should be triggered by the `run()` method of the Handler object. This reconfiguration can be done with the assistance of libraries that support the reconfiguration of distributed applications [KonPhD:2000].

Consequences:

+ Leads to a clear separation of concerns between the application functional code and the adaptation code. As a consequence, the resulting application becomes easier to design, implement, and maintain.
- The actions applied in reaction to an environmental change are hard-coded into the application and cannot be dynamically changed.
- Reflection adds complexity to the system which can make understanding and maintaining the system harder.

Implementation:

To implement this pattern, first it is necessary to make the event handlers capable of receiving event notifications by using a mechanism compatible with the notification mechanism chosen in the implementation of the **EVENT DETECTOR (3)**.

Then, it is required to implement handlers capable of changing the internal behaviour or structure of the application. Thus, normally the handler programmer must have a very good knowledge of the application implementation. One way to mitigate this requirement is to provide an interface in the application that programmers can use to set some parameters that determine how the application works. In this case, the application programmer would specify how the application could be configured (by specifiying which parameters can be set) and the event handler programmer would simply write the code that sets the proper values for the parameters.

Resulting Context:

By applying this pattern, the system is able to respond to changes in the environment by reconfiguring the applications, allowing for the implementation of self-adaptive applications. The set of reconfiguration actions to be taken are hard-coded and cannot be modified without recompiling the application. If the ability to dynamically redefiny the reconfiguration actions is desirable, then the **ADAPTIVE RECONFIGURATOR (5)** pattern should be used instead.

Related Patterns:

- The Reflection architectural pattern [Buschmann:1996] describes how to separate components of an adaptable system in a meta-level and a base-level. Base-level components deal with functional aspects of the system while meta-level components deal with non-functional aspects such as dynamic reconfiguration.
- Foote and Yoder [Foote:1995] describe some common reflective patterns that should be considered when building systems that need to be able to dynamically adapt at runtime.

Known Uses:

- The Video Datagram Protocol used by the Vosaic system [Chen:1996] uses a hard-coded adaptation algorithm that changes parameters of a video streaming session based on the monitored rate of dropped network packets.
- Chang and Karamcheti [Chang:2000] describe a framework for automatic configuration and run-time adaptation of distributed applications that hard-code alternative execution paths (algorithms) that are dynamically selected by guard expressions of control parameters.
- Noble and Satyanarayanan [Noble:1999] describes Odyssey, a platform for mobile data access. The adaptive application is divided in client and server components and the hard-coded adaptation mechanism allows to choose between different versions of the data being retrieved, so as to be compatible with the environment resource availability.

5. ADAPTIVE RECONFIGURATOR

Motivation:

An adaptive application must change its behavior in reaction to changes in its execution environment. The application can consider several different events that signal environmental changes; for each event, the adaptive actions that must be performed can vary. More flexibility can be achieved if the developer is allowed to specify collections of adaptive actions (adaptation policies) for each event, switching from one to another at run time depending on the application context. The adaptation mechanism can also evolve or be debugged without restarting the application if it allows dynamic loading and unloading of adaptation policies.

Problem:

How to vary at run time the collection of adaptive actions?

Forces:

- On the one hand, statically configuring the adaptation policies into the application code requires stopping, recompiling, and restarting the application whenever new code for an adaptation policy or changes in an old one are developed. These activities are infeasible for applications with high availability requirements. Dynamic loading and unloading of adaptation policies not only resolves this problem but also minimizes resource consumption, since only policies that are in use need to be loaded into the application code. This is particularly important if the computer on which the application is running has memory and processing limitations.
- On the other hand, the dynamic loading and unloading of adaptation policies implies application overhead.

Solution:

Decouple the Event Handler interface from its implementation by defining a uniform interface for each Event Handler that applies adaptive actions for a given environmental event. Develop one or more concrete Event Handlers that implement this interface. Each concrete Event Handler implements an adaptation policy. As illustrated in Figure 13, define for each application component a corresponding `Component Configurator` [KonPhD:2000]. The `Component Configurator` keeps track of the dynamic dependencies between the component and other system or application components and is also responsible for (1) disseminating events across inter-dependent components, whenever they affect several correlated components and (2) carrying out component-specific reconfiguration actions by calling component methods directly. Through this mechanism, the Event Handler can propagate the adaptive actions required to adapt the application to the new environmental state.

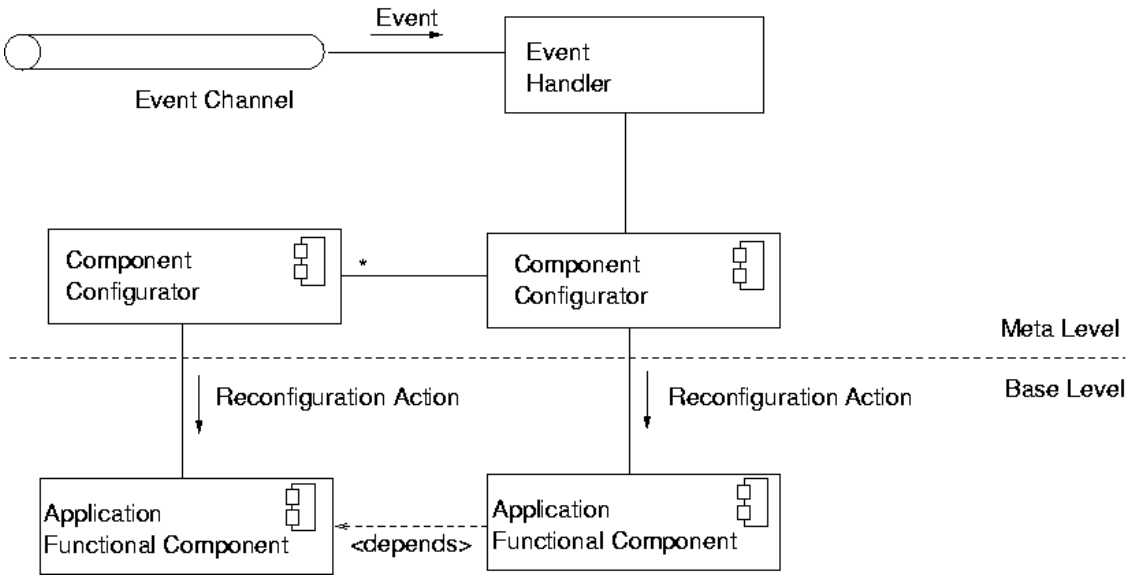

Figure 13 – Adaptive Reconfigurator structure

Example:

It is possible to design an application to be adaptable in ways that can be fully specified at design time, but it is difficult, if not impossible, to anticipate all the ways in which it may be required to adapt some applications. For instance, in mobile computing environments, the characteristics of the network connections can range from an inexpensive, very high bandwidth with low latency connection such as high-speed LAN, to a very expensive, low bandwidth with high latency connection such as GSM or infrared. Even the network address of the machine can change. Mobile applications should also be able to handle periods of disconnection. The application and data characteristics, and the user's context requirements and limitations may all change dynamically. Any of these contextual conditions can change without warning and to values unknown and unforseen by the application designer. Thus, it might be necessary to load new adaptation policies at runtime.

Consequences:

+ Several adaptation policies can be defined and reconfigured at run time for each environmental event.
+ The adaptation policies can be loaded and unloaded dynamically, allowing the adaptation mechanism to evolve or be debugged without restarting the application. This also minimizes resource consumption.
– The dynamic loading and unloading of adaptation policies generates overhead to the adaptation mechanism.
– The level of complexity increases making applications more difficult to develop and maintain.

Implementation:

Organize application components in two layers: (1) a metalevel layer, responsible for receiving event notifications describing changes on distributed resource usage and also

applying reconfiguration actions to adapt the application to the new environmental state; and (2) a base level, that provides the application functionality.

Define, for each application component, a corresponding `Component Configurator` object. As described in the Dynamic Dependence Manager pattern [Domingues, 2005], the `Component Configurator` keeps track of the dynamic dependencies between the component and other system or application components and helps to maintain runtime consistency in the presence of reconfigurations. `Component Configurators` are also responsible for disseminating events across inter-dependent components. Examples of common events are the failure or migration of a component, internal reconfiguration, or replacement of the component implementation. The rationale is that those events affect all the dependent components. This communication mechanism coordinates reconfiguration actions among the application components. The `Component Configurator` contains the code to deal with these configuration-related events. This approach provides a clear separation of concerns between the application functional code and the code that deals with the application reconfiguration.

As illustrated in Figure 14, create an `EventReceiver` that registers itself with the event channels (described in the **EVENT DETECTOR (3)**). The `EventReceiver` will be notified of events that indicate relevant environmental changes. Depending on the type of the event, it executes the appropriate actions required to adapt the application to the new environment state, using the `Component Configurators` to coordinate reconfiguration actions among the, possibly distributed, application components. Organize the classes that handle each environment event as a set of Event Handler strategies, using the Strategy design pattern. Figure 14 illustrates three strategies `(EventAHandler1, EventAHandler2, and EventAHandler3)` that can be triggered when an instance of `EventA` is notified.

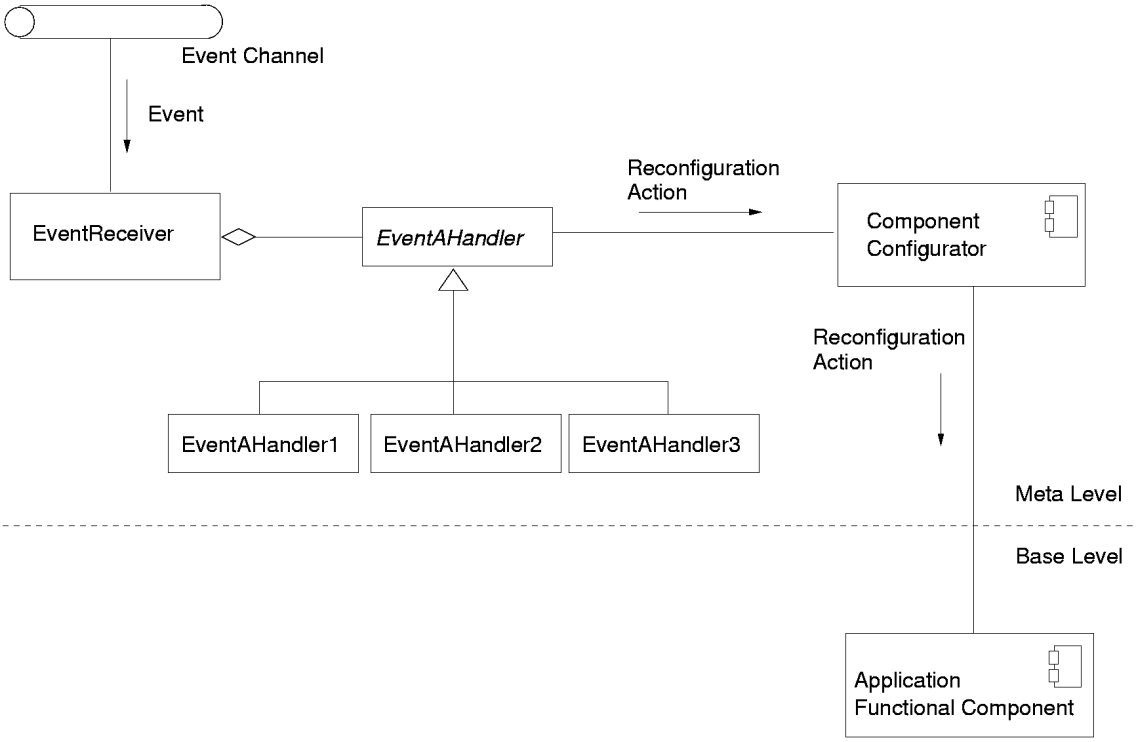

Figure 14 - Architecture for handling events and reconfiguring the application

Concrete Event Handlers should be packaged into a suitable unit of configuration that can be dynamically linked to the application, such as a dynamically loaded library (DLL) or a Java class file. The dynamic loading and unloading of Event Handlers can be controlled by specific configuration mechanisms such as the Component Configurator (forming a metametalevel, not illustrated in Figure 15). Other configuration operations, such as suspend and resume, can also be supplied. Suspending the execution of an Event Handler implies not executing the reconfiguration actions when the corresponding event is triggered. Resuming it, turns back the adaptive behavior of the application for the corresponding event.

Resulting Context:

By applying this pattern, the system is able to respond to changes in the environment by reconfiguring the applications, enabling the implementation of self-adaptive applications. In addition, the set of adaptation actions is also reconfigurable, allowing the application maintainer or operator to modify or add new reconfiguration strategies at runtime. This permits the construction of highly flexible and reconfigurable applications that can evolve and change radically at runtime without the need for shutdown and restart.

Related Patterns:

- The Strategy design pattern [Gamma:1994] explains how to build a system such that the algorithms it uses can be changed dynamically. The **ADAPTIVE RECONFIGURATOR (5)** can be implemented by enhancing an implementation of the **AUTOMATIC RECONFIGURATOR (4)** with the Strategy pattern.
- The Component Configurator pattern [Schmidt:2000] (not to be confused with the Component Configurator object [KonPhD:2000] used in the implementation section of the pattern described here) describes a mechanism for dynamically loading and configuring components into a running execution environment. This pattern can be used to load new Event Handlers dynamically allowing new forms of reconfiguring a system.
- The TypeSquare pattern described in Adaptive Object-Models (AOM) can be used for implementing the set of events that an extended ComponentConfigurator can handle [Yoder:2001; Yoder:2002]. Event definitions can be stored in a XML file that can be read at run-time in order to dynamically build the objects responsible for handling new defined events without the necessity of recompiling and restarting the extended ComponentConfigurator. AOM describes how to read the metadata file and dynamically build these objects using the Interpreter and Build patterns.

Known Uses:

- The Framework for Adaptive Distributed Systems [Silva:2003] allows dynamic loading new Component Configurators and new Event Handlers at runtime.
- The dynamicTAO reflective ORB [Kon&Roman:2000] allows dynamic loading new ORB components at runtime, including components that take care of the reconfiguration process itself.

Variant:

If the adaptation mechanism must provide more than one adaptation policy for a given event but dynamic loading and unloading its code is unnecessary, the Strategy pattern can be applied instead of the Component Configurator. Decouple the Event Handler interface from its implementation and develop concrete Event Handlers that implement the uniform interface. Each concrete Event Handler implements an adaptation policy and corresponds to a Concrete Strategy using the Strategy pattern terminology. A Context object must be configured with the concrete Handler to be used when the corresponding event is triggered. If the adaptation mechanism must switch from one adaptation policy to another at run time, all concrete Event Handlers must be instantiated as part of the application initialization. If only one policy will be used in a single application execution, it is only necessary to instantiate the Event Handler that corresponds to the policy that will be applied.

Pattern Language Summary

Computing environments today require systems to adapt quickly to changes, which often includes reconfiguring or adapting to an evolving environment. This paper presented a pattern language for assisting with this requirement, specifically with the problem of building automatically configurable and adaptive distributed systems. The pattern language outlines an architecture for describing "**When**" should an adaptation be done (monitors), "**What**" adaptation should be performed (event detection) and "**How**" to adapt the system (reconfigurators).

The **DISTRIBUTED MONITOR (1)** provides a simpler solution for monitoring distributed resources while the **ADAPTIVE MONITOR (2)** describes an extension that supports dynamic reconfiguration of the monitor. The approach uses rules for triggering events for when adaptations should be performed. The **EVENT DETECTOR (3)**, uses the rules and notifies the mechanisms responsible for the dynamic reconfiguration of the system. The reconfigurators provide a mechanism for actually adapting the system safely according to the specified rules. The **AUTOMATIC RECONFIGURATOR (4)** describes a simpler solution while the **ADAPTIVE RECONFIGURATOR (5)** describes dynamic reconfiguration of the reconfiguration process, thus making the reconfigurator more adaptable and reconfigurable.

There are orthogonal issues that will need to be addressed while applying these patterns such as Security, Fault-Tolerance, and Real-Time, which are beyond the scope of this pattern language.

Acknowledgments

The authors would like to thank Eugene Wallingford, Paulo Borba, Linda Rising, Giuliano Mega, and Eduardo Fernandez for their valuable thoughts and suggestions that greatly contributed to this work. We would also like to thank the Software Architecture Group from the University of Illinois at Urbana-Champaign and the SugarLoafPLoP'2005 Araucaria group for their valuable feedback and comments.

References

[BBN:2002] BBN Technologies. *QuO ToolKit User's Guide*, release 3.0.10, April 2002. http://quo.bbn.com.

[Buschmann:1996] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal, Peter Sommerlad, Michael Stal. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. John Wiley & Sons, 1996.

[Chang:2000] Fangzhe Chang and Vijay Karamcheti. Automatic conguration and run-time adaptation of distributed applications. In *Ninth IEEE International Symposium on High Performance Distributed Computing*, pp. 11-20, Pittsburg, Pennsylvania, August 2000.

[Chen:1996] Zhigang Chen and See-Mong Tan and Roy H. Campbell and Yongcheng Li. Real-Time Video and Audio in the World Wide Web. In *World Wide Web Journal*. 1(1). 1996.

[CORBA:2002] OMG - Object Management Group. *The Common Object Request Broker: Architecture and Specication*, November 2002. version 3.0.1.

[Domingues:2005] Helves Domingues and Marco A. S. Netto. *The Dynamic Dependence Manager Pattern*. Technical Report RT-MAC-2005-07, Department of Computer Science, University of São Paulo. 2005.

[Foote:1995] Brian Foote and Joseph W. Yoder Evolution, Architecture, and Metamorphosis. In *Second Conference on Patterns Languages of Programs (PLoP '95)*. Monticello, Illinois, September 1995. Also *Pattern Languages of Program Design 2* edited by John M. Vlissides, James O. Coplien, and Norman L. Kerth. Addison-Wesley, 1996.

[Foote:1998] Brian Foote and Joseph Yoder. *Metadata and Active Object-Models Collected papers from the PLoP '98 and EuroPLoP '98 Conference*, Technical Report WUCS-98-25, Department of Computer Science, Washington University, September 1998.

[Foster:1997] Ian Foster and Carl Kesselman. Globus: A Metacomputing Infrastructure Toolkit. In *International Journal of Supercomputer Applications*. 11(2), pp. 115-118. 1997.

[Gamma:1994] Erich Gamma, Richard Helm, John Vlissides, and Ralph Johnson. *Design Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley, 1994.

[Goldchleger:2003] Andrei Goldchleger, Fabio Kon, Alfredo Goldman, Marcelo Finger, and Germano Capistrano Bezerra. InteGrade: Object-Oriented Grid Middleware Leveraging Idle Computing Power of Desktop Machines.

In *Concurrency and Computation: Practice & Experience*. Vol. 16, pp. 449-459. March, 2004.

[Kircher:2004] Michael Kircher, Prashant Jain. *Pattern-Oriented Software Architecture, Patterns for Resource Management*. John Wiley & Sons, 2004.

[Kon&Roman:2000] Fabio Kon, Manuel Roman, Ping Liu, Jina Mao, Tomonori Yamane, Luiz Claudio Magalhães, and Roy H. Campbell. Monitoring, Security, and Dynamic Conguration with the dynamicTAO Reflective ORB. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'2000)*, number 1795 in LNCS, pages 121-143, New York, April 2000. Springer-Verlag.

[Kon:2000] Fabio Kon, Roy Campbell, M. Dennis Mickunas, Klara Nahrstedt, and Francisco J. Ballesteros. 2K: A Distributed Operating System for Dynamic Heterogeneous Environments. In *9th IEEE International Symposium on High Performance Distributed Computing*. Pittsburgh. August 1-4, 2000.

[KonPhD:2000] Fabio Kon. Automatic *Conguration of Component-Based Distributed Systems*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, May 2000.

[Kon:2005] Fabio Kon, Jeferson Roberto Marques, Tomonori Yamane, Roy H. Campbell, and M. Dennis Mickunas. Design, Implementation, and Performance of an Automatic Configuration Service for Distributed Component Systems. In *Software: Practice and Experience*, 35(7), pp. 667-703, May 2005.

[Maes:1987] Maes P. Concepts and experiments in computational reflection. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications Conference '87*, volume 22 of Sigplan Notices, pages 147-155. ACM, December 1987.

[Moreto:2001] Douglas Moreto and Markus Endler. Evaluating composite events using shared trees. In *IEEE Proceedings Software*, 2001. ISSN 1462-5970, 148(1).

[Moura:2002] Moura A, Ururahy C, Cerqueira R, and Rodriguez N. Dynamic support for distributed auto-adaptive applications. In *Proceedings of AOPDCS - Workshop on Aspect Oriented Programming for Distributed Computing Systems (held in conjunction with IEEE ICDCS 2002)*, pages 451-456, Vienna, Austria, July 2002.

[Noble:1999] B. D. Noble and M. Satyanarayanan. Experience with adaptive mobile applications in Odyssey. In *Mobile Networks and Applications*, 4(4):245-254, 1999. Kluwer.

[Schmidt:2000] Douglas Schmidt, Michael Stal, Hans Rohnert, Frank Buschmann. *Pattern-Oriented Software Architecture, Volume 2, Patterns for*

Concurrent and Networked Objects. John Wiley & Sonss, 2000.

[Silva:2003] Francisco J. S. Silva, Markus Endler, and Fabio Kon. Developing Adaptive Distributed Applications: a Framework Overview and Experimental Results. In *Proceedings of the International Symposium on Distributed Objects and Applications*. LNCS 2888, pp.1275-1291. Catania, Sicily, Italy, November, 2003.

[Sudame:1997] Sudame P and Badrinath B. *On providing support for protocol adaptation in mobile wireless networks*. Technical report, Department of Computer Science, Rutgers Universit, June 1997. http://www.cs.rutgers.edu/pub/technical-reports/dcstr-333.ps.Z.

[Vanegas:1998] Vanegas R, Zinky J, Loyall J, Karr D, Schantz R, and Bakken D. QuO's runtime support for quality of service in distributed objects. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98)*, The Lake District, England, September 1998.

[Welling:1997] Girish Welling and B. R. Badrinath. A framework for environment aware mobile applications. In *Proceedings of the 17th International Conference on Distributed Computing Systems (ICDCS'97)*, May 1997.

[Yoder:2001] Joseph Yoder, Federico Balguer and Ralph Johnson. Architecture and Design of Adaptive Object-Models. In *Proceedings of the 2001 Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'01)*. ACM SIGPLAN Notices, December 2001.

[Yoder:2002] Joseph Yoder and Ralph Johnson. The Adaptive Object-Model Architectural Style. In *Proceedings of the Workshop IEEE/IFIP Conference on Software Architecture 2002 (WICSA3 '02)* at the World Computer Congress in Montreal, August 2002. Software Architecture System Design, Development and Maintenance Edited by Jan Bosch, Morven Gentleman, Christine Hofmeister, and Juha Kuusela; Kluwer Academic Publishers 2002.

[Zinky:1997] John A. Zinky, David E. Bakken, and Richard E. Schantz. Architectural Support for Quality of Service for CORBA Objects. In *Theory and Practice of Object Systems*. April, 1997.