SYSTEM MODELLING AND RULE EVALUATION IN AN ADAPTIVE MOBILE OBJECT FRAMEWORK

LACHLAN DONALD

Supervisor: DR. CASPAR RYAN

Honours Thesis

School of Computer Science and Information Technology
RMIT University
Melbourne, AUSTRALIA

October, 2004

Abstract

Distributed applications are challenging to deploy on ad hoc networks of heterogeneous devices. This thesis proposes a methodology for providing mobile object frameworks with a model of the distributed application. The proposed model captures the runtime structure of the application, allowing the framework to make informed choices about how to adapt the application to changes in environmental conditions and network topology, as they occur. Using a set of predicate rules, the model can optimise the deployment of objects across the network of available devices, based on available resources and performance characteristics.

Contents

1 Introduction

Distributed applications are broken into logical components, or groups of components, and distributed amongst a network of devices (Emmerich, 2002). Traditional distributed applications, such as distributed databases, are decomposed into coarse components such as data persistence, user interfaces and computational elements. This decomposition occurs due to the coupling of the sub-components within each logical group. On a network of devices with similar capabilities, distributing these groups evenly across the available resources is a well understood area. (Ghosh, 1994) (Dewan et al., 1994).

Next Generation Internet (NGI) networks are composed of heterogenous devices connected over a variety of wireless and wired infrastructure (Moyer and Umar, 2001). These devices are often portable or hand-held and can exhibit a diverse range of performance characteristics, including memory capacity, processor speed, connectivity methods, screen size, input methods (e.g keyboard, stylus, speech), etc. Applications operating on these ad hoc networks must deal with a range of conditions, such as a constantly changing network topology and transient network errors. The changes in topology occur as portable devices move in and out of range of the network, or as devices are docked and undocked from cradles. As portable devices often maintain connectivity with wireless network connections, transient network errors occur frequently due to interference and physical obstacles that prevent clear signal transmission.

Deployment and configuration of distributed applications on these diverse networks becomes increasingly complicated (O'Sullivan, 1997). The added complexity stems from the inconsistencies between capabilities of devices, impeding even distribution of work between devices. For instance, attempting to distribute a processor-intensive task evenly between a desktop computer and a smart phone would not be practical, as the resources available on the phone are a fraction of those available on the desktop computer.

A class of distributed applications have appeared that operates on these ad hoc networks, known as mobile applications (Hawick and James, 2003). Mobile applications distribute components across a network of devices much in the same way as classic distributed applications, but the deployment of the components changes dynamically whilst the application is running. This characteristic allows mobile applications to accommodate for the varying capabilities of devices and for components to be re-located to devices where conditions are most favourable.

The advent of technologies such as Remote Method Invocation (RMI) (Waldo, 1998) has provided a means for distributing objects at an individual object level, with a high degree of transparency to the original application. By providing a more fine-grained mechanism for distributing individual objects, distributed applications are able to externalise the process whereby objects are defined as mobile and non-mobile. Capturing this deployment detail away from the core of the application into robust middleware layers, provides a means for breaking the strong coupling between the application and its deployment in a distributed environment. Using these techniques, a class of reflective middleware has evolved that can adapt to changes in environment (Schmidt, 2002) (Kon et al., 2002).

Adaptive middlewares consist of a chain of basic services that can be dynamically added or removed to facilitate change in the middleware. A service acts as a filter for communication between elements of a distributed application, allowing the middleware to adapt what is sent between nodes at runtime. For example, when bandwidth is constrained, a service might be dynamically added to the service chain, which would then compress all incoming and outgoing traffic. To facilitate mobile applications, a class of adaptive middlewares exist that can also modify the topology of the application transparently at runtime. These frameworks are known as mobile object frameworks (Jing et al., 1999).

In order to adapt to a changing environment, mobile object frameworks must maintain an aware-

ness of the topology of the distributed application as a whole, and be able to react to events that occur on the devices in real-time. This is accomplished by maintaining a model of the distributed system (Hong and Bauer, 1993), which captures a view of the application from the last point at which it was polled. The model is made up of symbolic representations of each device; a means of contacting the device, the mobile objects within them, and a collection of relevant performance metrics.

With the existence of a model, the entire network of devices can be monitored and components of the mobile application redistributed in response to changing environmental conditions in the surrounding system and in the objects themselves. By monitoring the state of the system as a whole, choices can be made about the most suitable locations for objects based on a set of predicate rules that are either dynamically generated or provided by the user (Weinsberg and Ben-Shaul, 2002).

A model that stores metrics and captures the relationships and locations of entities in the system when combined with predicate rules allows for flexible deployment across ad hoc networks and for runtime adaptation of distributed applications. This thesis proposes such a model, along with a language for expressing rules and a rule engine to evaluate these rules.

2 Literature Review

The system proposed in this paper builds upon techniques from a variety of areas. The following sections outline relevant research in the areas of mobile object frameworks, system modelling and rule formats, and how the research presented in this paper expands or builds upon them.

2.1 Mobile Object Frameworks

Mobile object frameworks provide some important infrastructure, such as the ability to transparently transfer an object between two devices on a network, and the ability to inspect distributed objects to determine their state. One such framework that provides these facilities is the MobJeX framework (Ryan and Westhorpe, 2004), which provides operational mobility facilities and resource monitoring. The MobJeX framework specifies the requirement for a rule-based controller, however it currently lacks a means for moving objects in response to changes in the network or application.

The MobJeX framework facilitates transparent object movement by creating proxies for each distributed object that provide an identical interface to their original non-distributed counterpart. RMI (Waldo, 1998) is then used to maintain stubs to the remote object within the proxy, allowing the object movement that is transparent to the application.

Other implementations of mobile object frameworks such as FarGo (Weinsberg and Ben-Shaul, 2002), JavaParty (Haumacher., 2003) and Proactive (Baude et al., 2000) exist, but they lack the workable implementations of transparent mobility. Additionally, the MobJeX project has been designed to co-exist with a central or de-centralized system of control, which makes it an excellent candidate for the system model and ruleset proposed in this paper.

The MobJeX framework provides a means for adapting non-distributed applications to operate on the often unpredictable topology of modern ad hoc networks of heterogenous devices. This capability is added at a source-code level with a pre-compilation process that augments Java classes with the ability to transparently migrate hosts. This approach allows for the object to be modified without affecting the rest of the application, and since the object runs on an unmodified Java Virtual Machine (JVM), the objects can be distributed to the majority of modern portable devices such as PDAs and Mobile Phones that provide a Java runtime.

The MobJeX framework specifies three separate conceptual subsystems; the mobility system,

which is responsible for moving the objects at a transport level, the system controller, which is the combination of a dynamic model and a language for expressing and evaluating rules, and an administration console that interacts with the system controller to affect change in the system. The MobJeX framework implements a simple model for viewing objects in the system via the administration console. This thesis expands upon this implementation by extending the capabilities of the model to include dynamic reflection of the underlying system and a declaratively ruleset for manipulating the model.

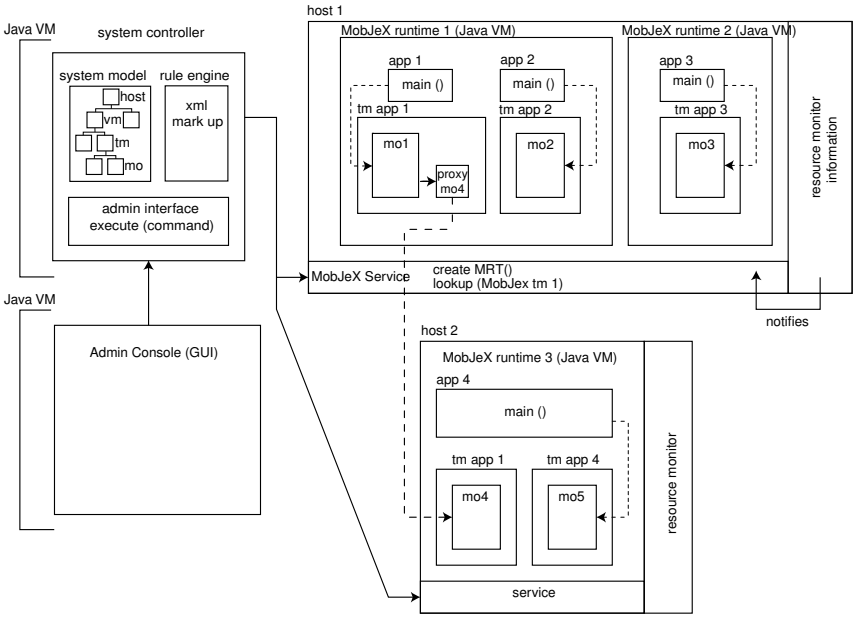

Figure 1: MobJeX Architecture

Figure 1 represents the architecture of MobJeX. The mobility subsystem is made up of a hierarchical collection of components. The highest level of component is a service, which embodies a physical host. The service contains runtimes which each represent a Java VM. The runtimes contain one transport manager per application running. A transport manager contains the actual mobile objects, known as mobjects.

Mobjects consist of a Proxy (Shapiro, 1986) object which each distributed copy of the application has. When called, the Proxy defers calls to the actual object, located remotely across a Network via Java's RMI (Waldo, 1998) mechanism. These objects are capable of transparently moving from local objects to remote objects without effecting the enclosing application.

Another key aspect of the MobJeX framework is that it is designed for ad hoc networks where bandwidth and latency can be major factors. One of the largest overheads encountered in an environment such as this is the time taken to perform remote RMI calls (Philippsen et al., 2000). Entities in the MobJeX system lookup other remote entities, either via a shared remote reference stored in an internal cache or, in the case of transport managers and mobjects, via a generated unique identifier. Objects in the MobJeX system all have unique identities, which allow for some caching techniques to be applied to object references to minimize unnecessary remote calls.

Highly coupled clusters of objects present a problem for mobile object frameworks, as the frequent calls between the objects cause large network overheads. One technique to avoid this problem is to group objects on hosts based on their levels of coupling. This ensures that highly coupled objects

will always be local to each other, allowing for optimisations to be made for local connections. The Pangaea framework (Spiegel, 2000) uses a technique called "static analysis" to determine how objects should be clustered. This technique processes the source code of an application, and builds a graph of connections between classes and methods. As this technique builds only on a static model it is inadequate for determining clustering in a dynamic mobile application. This thesis proposes a declarative ruleset for specifying locality constraints, which builds upon this technique to provide a solution for mobile applications.

2.2 Modelling and Simulation

Manipulating complex systems is a common application for computing. Large systems with many interacting components and interdependent events exist in all branches of science and are often too large to be visualised without the assistance of a model (Randell et al., 1999). Traditionally, models for complex systems are mathematical constructs defining initial system state, valid input and resultant state changes (Nassersharif et al., 1988). The advent of computing saw mathematical models like this extended to a larger scale, with the first software model being used to model the process of a nuclear detonation during World War II.

The process for deriving a model centres on analysis of the problem domain. The elements in the problem domain that are important to a model need to be captured, along with their relationships and constraints. This technique is known as meta-modelling (Sprinkle, 2004) as it embodies a common technique for deriving a model from a real system. This technique is used in a variety of modelling approaches, such as in the development of Integrated Development Environments such as Borland's JBuilder[1], or Eclipse[2]. Tools such as these allow developers to manage large development projects at a higher-level, by manipulating a model of the program they are writing. Tools such as Rational Rose[3] take this technique to a higher level of abstraction by allowing developers to build models of a variety of problem domains.

Computer Modelling falls into three categories, analog simulations (Malamos et al., 2002), discrete simulations (Watson and Lovelock, 1983) and agent-based (Gotts et al., 2003) simulations. The following section outlines each category briefly and describes its relevance to Distributed Applications.

Analog simulations are mathematical models that use either partial or ordinary differential equations to define the behaviour of the system. The equations are periodically recalculated and the result used to cause state change in the system. Analog simulations are typically used to model complex physics such as fluid dynamics (Schulz et al., 1999), or in analysing situations that would be difficult to reconstruct in a lab, such as the physics of bird flight (Wu and Popovi, 2003).

Representing the behaviour of distributed applications in a series of equations is often effective for analysing overall system behaviour (Malamos et al., 2002), but in the case of mobile object frameworks the requirement is for modelling on an object level and the optimum configuration for each object rather than the emergent patterns of system behaviour.

Discrete simulations are composed of a queue of events. The model interprets these events (generally in real-time) and triggers new events in response to conditions it detects. Models that can't be represented with differential equations, or models where events occur probabilistically, are generally represented in this fashion. Models such as these are popular for modelling event occurrences and reactions. An example is modelling environmental events, such as the Daisyworld

[1] http://www.borland.com/jbuilder/

[2] http://www.eclipse.org/

[3] http://www.rational.com

(Watson and Lovelock, 1983) model which modelled a planet populated by only two species of daisies.

Discrete Simulations can be used to simulate distributed applications, as state-change in the system is effected by events. The linear processing of these events and their application to a system model represents the basic behaviour desired from a dynamic system model.

A specialized type of Discrete Simulation exists that extends upon the classic approach of maintaining event queues, by allowing the interaction between entities to affect state change. This technique is called Agent-Based (Gotts et al., 2003) simulation and is used for formally expressing discrete, interacting components where behaviour cannot be assumed constant over similar components, such as when modelling social interaction. In Agent-Based simulations, each component stores state and interacts with other components. In the realm of mobile applications, this technique allows interactions between distributed objects to be captured as events. As described in Section 2.3, events can be used to trigger rules.

Distributed applications such as active databases have long used agent-oriented techniques for modelling static and dynamic relationships between objects and for ensuring constraints are enforced when the database state changes (Cornelio and Navathe, 1993) (Beeri and Milo, 1991). The agent-oriented models enrich the realm of active databases by allowing the database to manage an entire domain of objects and maintain active rules that can trigger change at an object-based level. This is advantageous for databases that are distributed, as constant remote connections between peering database nodes for the purposes of maintaining relationships requires high network overheads.

For complex distributed systems, maintaining a model of the entire domain allows for the inter-object relationships to be monitored and managed. Models provide the requisite information about the application required to support rules which are triggered by events in the model, such as in Kappel and Retschitzegger's work (Kappel and Retschitzegger, 1998).

2.3 Rule Formats

Constraints such as object co-locality (as discussed in Section 2.1) cannot always be adequately captured by static analysis of the application (Spiegel, 2000). Providing a language that allows for the specification of actions to be performed when a set of conditions are met, allows users to tailor the deployment of mobile applications to the specific needs of the application and the environment that the application is being deployed in.

Chan and Chuang propose a reflective middleware, Mobipads (Chan et al., 2003) that provides a rule format for service-oriented contextual adaptation. The rule format specifies a number of 'composite events', which are sets of predicates that match environmental conditions corresponding to certain environmental situations, for example 'High Bandwidth' or 'Low CPU Availability'. Mobipads allows for these composite events to trigger changes to the chain of services that process all incoming and outgoing communications, allowing on-the-fly changes such as re-compression of images or the conversion of rich media files to text files for low-powered devices. Services for common operations such as this are provided, but most applications will require specialized services to be written and compiled into the middleware.

In contrast to the approach of adapting the communications layers used by the Mobipads framework, the MobJeX framework adapts the application to the environment by moving objects to the most appropriate host. A ruleset is proposed in Section 5 that uses an event-based rule language to cause this movement in the MobJeX framework.

Event-triggered rulesets are also commonly used to drive database systems, such as the work of Yoo, Kim and Chai (Yoo et al., 1999), in which a ruleset is proposed that monitors integrity constraints in a distributed database. Sets of rules are maintained by the system and as distributed records are

changed, events are triggered. Rules may be entered dynamically to make changes to the system while it runs. A ruleset manager is used to monitor the ruleset itself, ensuring that rules are compiled into an executable form if they change and also calculating which rules trigger other rules, so a 'trigger graph' may be built up. The trigger graph is used to ensure that constraints can't cause the system to deadlock and also to allow the rule manager to plan in which order to execute rules efficiently.

Rulesets for databases are typically domain-specific; hence an abstraction of their techniques is required. Adi and Etzion (Adi and Etzion, 2004) propose a general-purpose rule language and runtime called Amit for monitoring composite events using the ECA: Event-Condition-Action paradigm. Scenarios are described in the XML-based ruleset language, which detail a series of events that must occur, possibly over a span of time. Events are divided into two categories, external events and internal events. External Events are generated by external entities, for instance in a mobile object framework, an external event might be a node joining the network. Internal events are generated as the result of attributes of entities within the framework changing, for instance when memory is low and CPU utilisation is high on a device, an internal event may be fired that indicates that resources are low and that objects on the device are candidates for movement.

The Amit framework(Adi and Etzion, 2004) breaks event detection and rule processing into three stages; collection, detection and consumption. The collection stage is where events are gathered from external sources. The detection stage is where events are sorted, based on their relevance to scenarios detailed in the ruleset. The consumption phase is where instances are removed from the event queue and processed. Several useful data structures are proposed for storing rules and events, which could provide the basis of a rule engine.

The FarGo (Ben-Shaul et al., 1999) object mobility framework describes a scripting language for responding to events in the system. Figure 2 shows a sample of the language. The proposed scripting language uses 'Event-Action' pairs in the form `on event [at core] do actions`. In FarGo a core is the runtime environment, such as the MobJeX runtime described in Section 2.1. This syntax allows simple scripts to be triggered by a set of defined events. The events depicted in Figure 2 are `completDeparture` and `coreShutdown`. `completDeparture` is triggered when a mobile object cluster (known as a '*complet*') is moved, and causes the complet to be added to the user interface and then moved to the the 'best-fit' host. The action `coreShutdown` is triggered when a core shuts down. In the second rule, this event is triggered when a particular host shutdown occurs, causing all its '*complet*' components to move to a specific host.

```
1  Script EngineScript {
2     on completDeparture(complet, target) do {
3        engine = thisComplet();
4        userInterfaces += complet;
5        best = Locator.findBest(userInterfaces);
6        move(engine, best);
7     }
8     on coreShutdown(coreName) do (
9        moveAll("zeus.tecb.nion.ac.il");
10    }
11 }
```

Figure 2: FarGo Scripting Language (Weinsberg and Ben-Shaul, 2002, page 168)

The FarGo scripting language was not implemented, but demonstrates how a rule language could be used to add adaptive rules to a mobile object framework. Some obvious limitations with this particular syntax are the need for a compiler or interpreter for the custom syntax and the need for programming expertise to create scripts. A more fundamental limitation is the restriction imposed by

limiting triggering events to a small set of predefined external events, such as object movement or host creation or shutdown. This eliminates the possibility of the scripting language being used to optimise object's locations based on environmental attributes, such as available processing resources or current system load. The rule syntax proposed in Section 5 allows for complex events to be specified by the user.

3 Research Questions

The previous sections have described system modelling, rule formats and mobile object frameworks. In particular, the architecture of the MobJeX framework was reviewed and the need for a dynamic system model which can map the topology of mobile applications was described. Deploying these applications on an ad hoc network of heterogenous devices requires a means for adaptation. A proposed technique for adaptation is to provide the mobile object framework with a language which specifies 'event-action' rules that respond to conditions in the system model. This leads to a number of key research questions:

1. **How should the system model be represented?**

   Issues under consideration are, which elements are appropriate for modelling the MobJeX system, what interactions exist between these entities and how can these entities can best be arranged to accurately reflect the runtime characteristics of the system.

2. **What is the performance of such a representation?**

   Model schemas may have varying performance characteristics, therefore this thesis will quantify the scalability, fault-tolerance, performance and security of the proposed system model representation.

3. **How should the ruleset be specified?**

   Many language semantics and syntaxes are possible. Therefore, the most appropriate ruleset for the expression of 'event-action' rules for the model should be selected, based on the stated requirements of the MobJeX framework.

4. **How should the ruleset be executed against the model?**

   Different rulesets can be evaluated using a variety of techniques. Answering this question will involve investigating various techniques and evalutating the relative performance of the chosen solution.

4 Modelling Object Frameworks

This section proposes a dynamic model for mobile object frameworks which is capable of effectively modelling the domain of distributed systems, specifically the MobJeX framework as described in the Literature Review in Section 2.1. The model stores the topology of the system along with metrics that reflect the current state of the mobjects, including pending operations such as movement and creation. This information provides a foundation for the declarative rules discussed in Section 5.

4.1 Background / Requirements

Creating a dynamic model first requires that the problem domain be carefully analysed. Analysis and the formulation of a model is performed with two specific quantities in mind, abstraction and perspective (Benjamin et al., 1998). The level of abstraction of a model relates directly to the quantity of elements, or the 'level of detail'. Within a particular level of abstraction, perspective (Law and Kelton, 1997) is the particular set of elements are relevant to the model.

Determining the correct level of abstraction can be approached with an iterative process of trial-and-error (Law and Kelton, 1997) whereby the highest workable level of abstraction is captured and then individual elements are refined down to more specific levels of abstraction. This process is applied to mobile object frameworks by starting with the abstract concept of a distributed application and then refining the abstraction down to the level of the chosen implementation, the MobJeX framework.

4.2 Model Architecture

The dynamic model was created by capturing all of the key abstractions present in the MobJeX mobility system (described in Section 2.1) as model classes. As the Literature Review details, the Mobility Subsystem is made up of hierarchical containers of entities, starting at the coarsest grain with individual hosts and progressing down to the finest-grain, mobjects.

The primary model entities form a classic design pattern, Composite (Gamma et al., 2003). This pattern applies to domains where objects form a hierarchical structure and it provides a common interface for navigating up and down the hierarchy. This technique was used for the primary model entities to allow them to inherit basic container operations such as the ability to add, remove and list children entities from a single parent class. Additionally traversal of the entire tree is simplified, as traversing classes can use a single means of navigation, for instance an 'Iterator' or 'Visitor' pattern (Gamma et al., 2003).

Figure 3 and Figure 4 details an architecture for the dynamic model. This provides a framework that represents the entities in the MobJeX framework and the relationships between them, in order to address Research Question 1.

Instead of providing a single interface for entities that are containers as well as model entities, the responsibilities were split into two interfaces, `ModelEntity` and `ModelEntityContainer`, shown in Figure 3. A `ModelEntity` is any entity which can be identified and has a parent container. For each `ModelEntity` there exists a corresponding physical object in the MobJeX framework. A `ModelEntityContainer` is a container for `ModelEntity` objects. Splitting the functional roles allows for the top-level container, `Model`, to be only a `ModelEntityContainer`. This prevents accidental attempts to move the model to another location and also deals with the problem that a model as a whole can't be uniquely identified.

The `Observable` interface is provided as a means for other objects to subscribe to an individual `ModelEntity` to receive updates on changes to the entity. This provides a mechanism for other elements of the system such as the rule engine discussed in Section 5 to perform actions when the metrics of an object change. The `Model` itself can also be subscribed to by this mechanism and it provides notices when a `ModelEntity` is created, destroyed or moved.

Operations on the model were defined using the 'Command' pattern (Gamma et al., 2003). The intent of this pattern is to encapsulate a logical operation in an object, which can later be executed without an intimate knowledge of the internals of the command. This approach decouples the rest of the system from directly modifying the model. To apply system specific behaviour, when a `Command` is received by the MobJeX system controller, it wraps the command in a system specific command

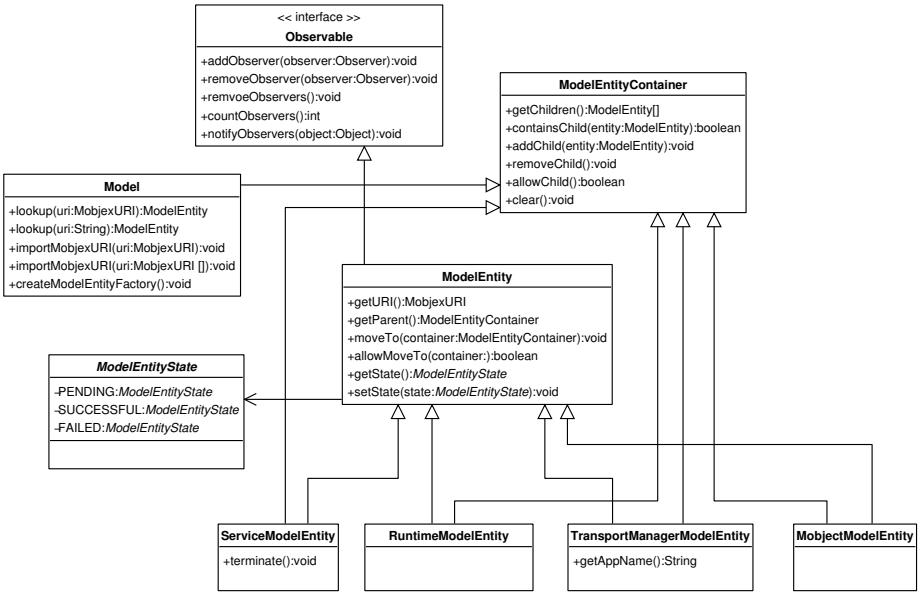

Figure 3: Model Architecture (UML 2.0)

(known as a Decorator (Gamma et al., 2003)).

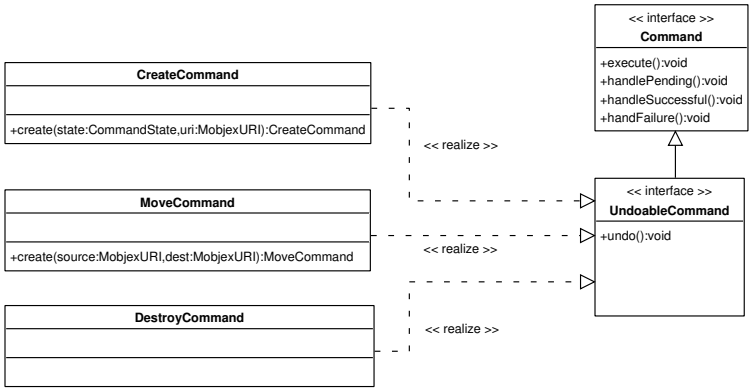

Figure 4: Semantic Commands (UML 2.0)

Figure 4 shows the three fundamental command types that the model requires, `CreateCommand` which is responsible for creating new `ModelEntity` objects, `MoveCommand` which moves a mobile object from one location to another and `DestroyCommand` which terminates an object and removes it from the model.

Commands further facilitate synchronization between separate models, such as when the model was distributed. Whenever one portion of the system makes a change, such as creation of an object, a command is sent reflecting the change to the other portions of the system. The other portions of the system in turn apply the command locally and send it on to further portions of the system that they are

connected to. Because each separate system applies its own specific behaviour as a wrapper around the commands it receives, the original command is not affected and can be forwarded in the same state as it was received. This means that the administration console can create a command, apply it to a local version of the model that is used to generate the GUI and then send that same command asynchronously to the System Controller. This asynchronous mechanism allows for instantaneous feedback for the user of the GUI, as changes can be represented locally in parallel to the command executing remotely.

Isolating the point at which domain-specific logic is applied to the model allows for the model to have a high degree of reuse. The model was designed to have fragments distributed across a network and to encapsulate access to the MobJeX subsystem, however its modular nature allowed it to be used in other realms of the MobJeX system in ways which had not been foreseen. The administration console uses the model to supply information to the GUI and the MobJeX service uses a read-only model for fast lookups of stubs and efficient traversal.

Providing a dynamic model of a distributed application allows for certain concepts to be overlayed that would not be possible to implement at an operational level. One such example is a scheme for uniquely identifying and locating objects. This technique is used within the model to decouple commands from actual model entities. Additionally it makes it easier to write rules that refer to specific objects within the system.

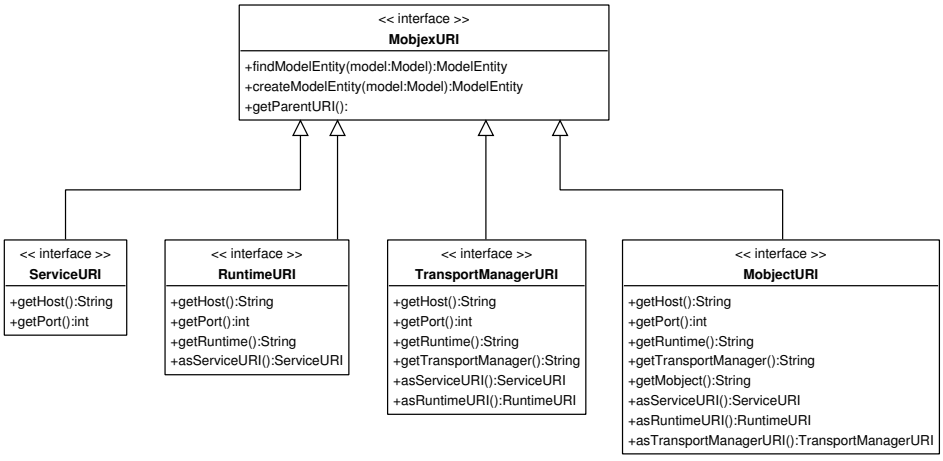

Figure 5: Identity Classes (UML 2.0)

Berners-Lee defines a scheme for encoding unique identifiers for locating resources, known as Universal Resource Identifiers (URI) (Berners-Lee et al., 1998). URIs are made up of a string of slash-delimited tokens that combine to provide a unique identifier for a particular entity, along with information about the protocol over which it operates. Figure 5 shows the model entities used to represent the URIs and shows how URIs exist in a hierarchy, whereby a lower URI type can have the higher types extracted from it. A syntax was created for MobJeX URIs that identifies entities in a MobJeX mobile application. Each level, or fragment of the URI indicates a portion of the address of the entity. As a whole, the URI can uniquely identify the associated entity. Every entity in a model has a URI as described in Figure 6.

The *hostname* is the TCP/IP hostname of the device on which the service is located, the *service-*

```
mobjex://hostname:service-port/runtime-id/tm-id/mobject-id
```

Figure 6: The Mobjex URI format

port is the port that the Service is running, the *runtime-id* uniquely identifies the MobJeX runtime, The *tm-id* is the unique identifier of the transport manager and the *mobject-id* is a unique identifier assigned to the mobile object.

As URIs provide a consistent means for identifying model entities, they also serve as index keys for the caching mechanisms distributed through out the MobJeX system. Further work on this topic might include optimising this process by creating an indexed data structure that exploited the hierarchical nature of the URIs. In the current implementation the primary use of this property is in the model, where a cache of recently looked-up model entities is kept.

Implementation of the dynamic model and the controller is based around the requirements of the MobJeX framework as described in Section 2.1; scalable, asynchronous, fault-tolerant and secure. The following sections list of these requirements, how each was implemented and how the resulting implementation was tested. Unless otherwise noted the solutions proposed are original, with citations provided where previous work is referenced.

4.3 Scalable

Modelling large networks of constantly changing devices requires a model that can scale as the network increases in size and changes. This requirement necessitates a technique where portions of the model are dynamically distributed to well-suited devices as they become available. The ability to dynamically adapt to changes in topology and device capabilities is a key requirement for the model, as the utility of a scalable mobile object framework becomes severely diminished if its system of control restricts the ability to scale.

4.3.1 Implementation

The implementation addressed scalability by ensuring that the model could be broken into smaller pieces and distributed in much the same way that the mobile application is distributed. Subsections of the model can be distributed to different locations and processing of rules deferred to those locations. This functionality is achieved by the structure of Command objects described in Section 4.2.

As the changes to the models are made via the commands, disconnected remote models can remain synchronized using a publisher/subscriber technique (Eugster et al., 2003), whereby models register with other models as subscribers and then when a model receives a command, it executes it locally and passes it on to its subscribers. This creates a network of interconnected models that can make changes to their local model and have the changes reflected in the remote peers.

To ensure that commands aren't propagated in cycles, a central controller would be used as a command router or Mediator (Gamma et al., 2003). This would allow the other controllers to be decoupled from each other. This decoupling allows for different routing strategies to be dynamically chosen. In the present implementation the routing controller is a fixed address controller, but future implementations might improve on this by holding "elections" within a cluster of controllers to elect the most suitable controller (Kim et al., 1995). This solution provides the benefits of a central controller for routing whilst retaining the fail-over properties that are beneficial in a ad hoc network.

4.3.2 Testing

A number of scalability tests were performed on the model to determine its ability to cope with large volumes of objects and commands. Commands are executed against models with varying quantities of objects. As the model structure is hierarchical, the distribution of the total amount of objects throughout the different levels dictates variables such as the time taken for lookup, insertion and deletion operations.

The tests were performed on a Pentium 4 2.8Ghz PC with 1Gb DDR-RAM running Windows XP build 2600. During testing, the workstation was disconnected from the network and all user-space processes were disabled. Tests were executed in Sun's Java 1.4.2_05 JVM with a 256Mb initial heap size to minimize delays due to garbage collection. Incremental Hotspot optimisation was also enabled to reduce spikes in test times caused by intermittent virtual machine activity.

Timing was performed using the Java Library *hrtlib*[4] which is implemented as a Java Native Interface (JNI) wrapper around the Windows *QueryPerformanceFrequency* and *QueryPerformanceCounter* methods. Using this technique as opposed to the Java API's *System.currentTimeMillis()* allows for much finer timer resolution, in the realm of sub-10 microsecond increments.

The test aims to measure the response times of the model under increasing volumes of commands and model load. This is accomplished by first setting up a basic model with a service, runtime and transport manager. 10,000 mobjects are then created, each with a *CreateCommand*. The time taken to execute each command is measured and recorded to a text file.

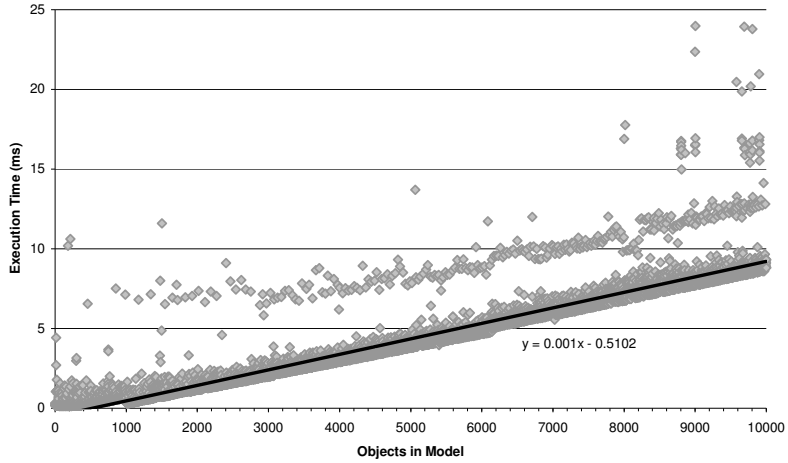

Figure 7: Creation Times for Model Entities

Figure 7 and Figure 8 show that the trend line for execution of a *CreateCommand* and a *MoveCommand* increase in a linear fashion. The outlying data points occur due to random occurances during the tests, such as unpredictable CPU scheduling or I/O delays. The linear complexity is made possible by the use of internal hashtables that store the entities. Profiling has shown that a large component of the time taken for commands to be executed is taken up with looking up the model entities in the model that correspond to the provided URIs, as described in Section 4.2. This is demonstrated by Figure 9.

Further tests were conducted where the quantity of services, runtimes and transport managers were varied, with the 10,000 mobjects being distributed to random transport manager's. The results of these tests showed a gradient of the same order of complexity as the gradient in Figure 7 and Figure

[4]http://www.javaworld.com/javaworld/javaqa/2003-01/01-qa-0110-timing.html

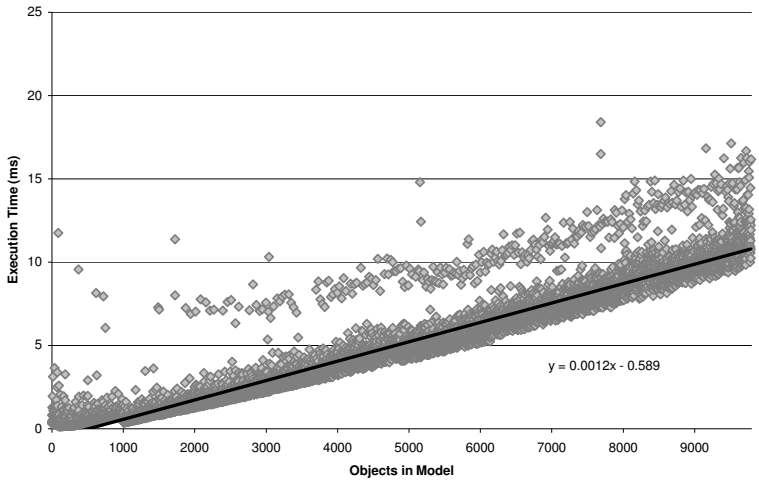

Figure 8: Movement Times for Model Entities

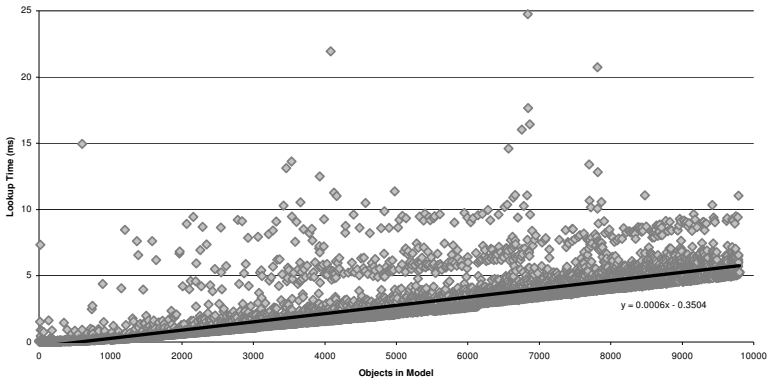

Figure 9: Lookup Times

8. This result was due to the fact that, regardless of the distribution of services and transport manages, the lookup times were of a linear complexity, and the same lookup strategies were employed in both tests. For the sake of brevity the results have been omitted.

The linear performance characteristics of the previous tests indicate that the simple model (as described in Section 4.2) provides a scalable model representation, relating to Research Question 2.

4.4 Asynchronous

In a distributed system, particularly one running on an ad hoc network, the speed at which commands execute can be unpredictable. Speeds can range from extremely slow wireless connections, such as those seen in a network of portable devices, to extremely fast connections such as those available in grid computing. Executing commands synchronously in this sort of environment introduces potential 'liveness' problems into the system, whereby certain commands might occupy the system for much longer than others, causing other pending commands to 'starve'. As commands return no value, executing them asynchronously provides a pragmatic approach to increasing the concurrancy of the system.

As described in Section 4.3, portions of the model are distributed and remotely synchronized by sending Commands back and forth. To provide feedback to the local system as quickly as possible, asynchronous commands must first update the local model before being sent off to remote models. This is most immediately obvious in the Administration GUI (described in Section 2.1), or in any user-facing component of the system and provides a measure of compromise between a classical synchronous system and a completely time-insensitive messaging system, such as Java's JMS framework (Terry and Shawn, 2002).

4.4.1 Implementation

Commands must have an associated state which indicates whether the command has been completed or if the change it reflected in the model is still pending at some remote site. In the initial implementation, model entities can have one of three states: 'pending', 'completed' and 'failed'. These states are specified in a command, and when the command executes the state specific changes are applied to the affected portions of model.

This has the advantage that rules can take into account actions as soon as they happen rather than waiting for the change to execute. An example of where this is beneficial is in a situation where mobjects are moved to another host with more available resources. Until the mobility subsystem executes these changes, the system model records these mobjects as 'pending'. Further load-balancing rules can take the existance of these 'pending' entities into account to prevent the system from overloading the host.

Until model elements are in a completed state they cannot be the subjects of further commands. As described in Section 4.5.1, a failed command will cause the system to roll back the recent unsuccessful changes to the model.

4.4.2 Testing

For the test, a varying number of system controllers, each with an separate model are chained together in a "daisy-chain" configuration (as shown in Figure 10) whereby each controller subscribes to the commands sent by the one before it and publishes commands to the one following it. The test application acts as a transmitter for commands at one end and as a terminator at the other. Using this

technique, asynchronous commands can be counted as soon as they have departed the testing application. Synchronous commands can be emulated by only sending a new command from the testing application when the terminator receives the previous command.

The testing environment was the same as described in Section 4.3.2. Tests were run multiple times and the values shown are the average of the runs. The tests carried out aim to measure the speed at which the model will accept commands using asynchronous and synchronous commands. This test was conducted by constructing a mock command object that executes for exactly 5 milliseconds, so as to factor out the time taken to interact with the model. This allows for the infrastructure over which commands are executed to be tested. Figure 11 shows the command throughput rate, in terms of the amount of commands executed per second. This rate was recorded over a period of thirty seconds.

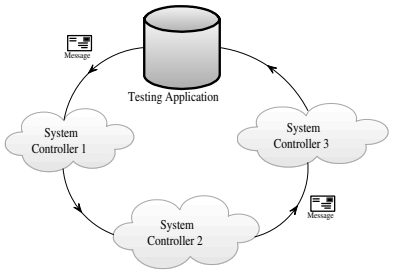

Figure 10: Testing Component Layout

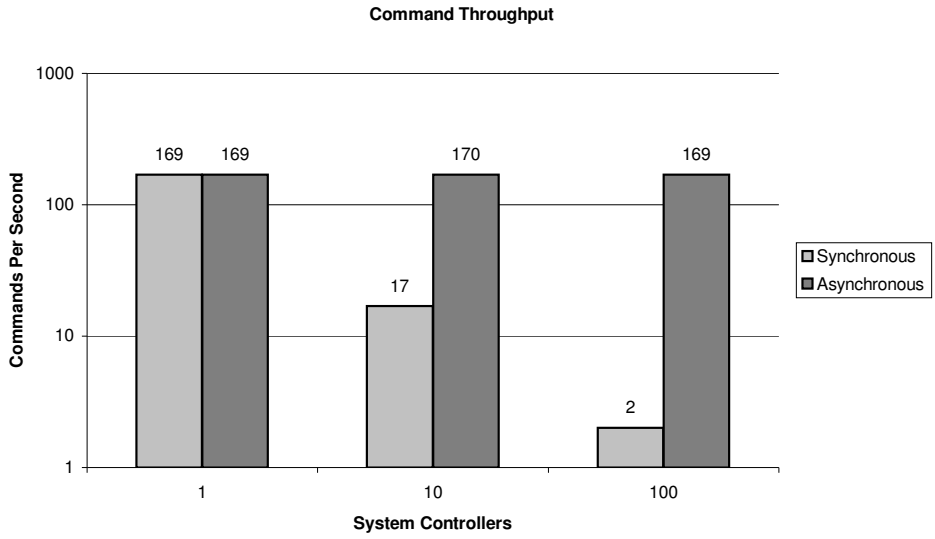

Figure 11: Comparison of Call Methods

The results show that both approaches are comparable in speed in a single controller environment. Environments with more than one controller, such as in a realistic MobJeX environment, have a consistent throughput rate when asynchronous semantics are used, whereas with synchronous semantics the throughput rate declines sharply as the number of controllers grow. This result addresses Research Question 2 as it shows that asynchronous call semantics provide an efficient, scalable approach for interacting with the MobJeX framework.

4.5 Fault-Tolerant

As discussed in the Literature Review in Section 2.1, mobile applications run on ad hoc networks and must adapt to changing network topology and provide a mechanism for recovering from transient network errors.

4.5.1 Implementation

When a device in a cluster of MobJeX-enabled devices leaves the network, portions of the application which were on that device become unavailable to the rest of the network. A technique for mitigating this problem is to capture the state of the system regularly, using a technique called checkpointing (Elnozahy et al., 2002).

Capturing the state of the entire system is a potentially expensive procedure; due to the fact that the model contains large amounts of information about the system it models, such as metrics and remote stubs for individual mobjects. To minimize the amount of information that must be stored in each image of the system, incremental checkpointing (Palaniswamy and Wilsey, 1993) is used. Rather than saving the entire state of the system, the points of change for the model are monitored and the actions that cause state change in the model are captured.

In the model, commands represent semantic actions on the model, or in simpler terms, actions that cause change in the system. The infrastructure used for sending and receiving commands provides an excellent point for logging and the introduction of additional commands. Recording state snapshots then becomes a matter of writing a log of commands executed to disk and when an error occurs reading the commands from disk and replaying them. This additionally allows for the state of the system to be "rolled back" to a particular point in the past.

As the errors that occur in ad hoc networks can be transient, the model must provide a variety of different error handling strategies. A pragmatic approach to recovering from errors that appear to be connection-level is to simply replay the failing command a number of times with a delay in between until the error either disappears or it can be concluded that the error is long-term. In this situation a 'failed' command can be sent back to the sender and in due course the user notified. Alternately, the command can be saved and re-tried at a later point in time.

Another benefit of recording model state change in atomic commands is that a layer of caching can be added to the model that groups commands sent between remote portions of the model in aggregate commands which are transparently unbundled on the receiving end. Whilst this technique incurs the overhead of caching the commands in memory and then bundling them in a new command, performance benefits are found on slow networks, where remote connection times are the limiting factor.

4.6 Secure

Security is a large topic, especially for a distributed system. A full analysis of the security implications of the interactions between the subsystems of the MobJeX framework, specifically the model and rulesets, would provide a good point for further research and whilst a comprehensive solution is beyond the scope of this paper, several techniques are proposed which employ 'good-sense' measures to reduce the risks inherent to distributed applications.

The primary point of weakness in the dynamic model is the communication lines. Models communicate with other models by sending executable commands, so the possibility exists for an attacker to simply register a rogue model and execute commands that would compromise the system. To prevent this, the system needs to employ techniques to prevent messages being intercepted, played back

out of sequence and also to confirm the identity of the sender. These objectives can be met by using public key cryptography.

Distributed portions of the model could exchange public keys (Halevi and Krawczyk, 1998) as part of their initial publisher/subscriber process and then messages could be signed and encrypted before being sent. Messages Authentication Digests could be included with the commands such that out of sequence commands inserted as part of a replay attack could be detected (Oehler and Glenn, 1997).

5 Rule Formats

This section proposes a set of predicate rules that express constraints that must be maintained and actions to perform as a reaction to events in the model. These predicate rules (hereafter referred to as a ruleset) provide a means to specify application-specific constraints, such as specific resource requirements or the need for object co-locality. In addition to the specification of a ruleset, the design and implementation of a rule engine, which is capable of evaluating the proposed ruleset is discussed.

Formats for rules can take on a number of different forms. However at the core of any syntax or grammar used to express a set of rules lies a set of key requirements, or use cases, that specify the distinct goals that the ruleset must be able to accomplish. The following section outlines the requirements for rules and the rule engine that executes them. The subsequent sections describe a set of semantics and syntax to answer Research Question 3. An implementation of a rule evaluation engine is described to answer Research Question 4, along with tests to demonstrate the integration of the rule syntax into the MobJeX framework.

5.1 Use Cases

A distributed system is optimised by moving objects to locations that have the best match of performance characteristics, for instance a large object would require a host with large amounts of free memory, whereas an object that made frequent network calls would require a host with high-bandwidth and low network utilization. The following use cases represent the requirements for a ruleset used to manipulate the MobJeX framework as derived from the goals (Ryan and Westhorpe, 2004) of the MobJeX project, as captured in Sections 4.3 to 4.6.

1. **Query a single mobject for metadata**

   As stated in Section 4.2, the model contains objects that represent entities in the mobile object framework. The model entities contain information about the remote object, such as performance metrics, details of the object's past movements and metadata such as the objects creator, purpose, etc. The rule engine must be able to query the model to retrieve this information and in turn make it available to the ruleset.

2. **Query a variety of mobject containers for one that meets a set of conditions**

   Querying a hierarchy of objects for one that meets a set of environmental parameters enables the ruleset to specify candidates for actions without specifying a fixed object. This is crucial for operations such as move, where the destination is potentially unknown.

3. **Perform an action when condition(s) are met**

   This use case provides the basic reactive functionality of the ruleset, providing a means to react to environmental conditions or errors, for example when the resources of a host are low, an action can be triggered to remedy the problem.

(a) **Move a mobject**

Providing a ruleset action to move an object when it meets or exceeds specified conditions allows for a rules to be created that perform context-sensitive optimisation and load-balancing. For example, moving large objects to targets with greater amounts of free memory.

(b) **Log a message**

Often users will want to monitor a system for certain critical conditions without actually reacting to them. Providing a means for logging a message facilitates this process. Potentially this might be implemented as a system log message or in the form of an email.

4. **Respond to error conditions**

Due to of the asynchronous nature of distributed systems, errors can arrive long after a command was sent. The rule format must allow for the capture and handling of error messages.

(a) **Respond to error by retrying the command**

As described in Section 4.5 certain errors are transient and can be corrected from simply by retrying the erroneous command. The ruleset must be able to specify that a command be retried if a particular error occurs. Providing a mechanism for bounding the number or retries would prevent a terminal error, that was detected as transient, being infinitely re-tried.

(b) **Respond to error by trying an alternate command**

Often deployment situations will dictate that if a command fails an alternative command should be executed. For instance, if moving an object fails a number of times a message could be logged, or a new object created.

5. **Create a co-locality constraint between n objects**

In a distributed application, high coupling in clusters of objects can result in high network latency due to the frequency of remote method calls. A strategy to minimize this problem is to ensure that the objects that are highly coupled are located on the same host so that the mobile object framework can optimise remote calls. Providing a rule that allows objects to be constrained to a group allows for the user to control this behaviour.

5.2 Semantics

Selecting appropriate semantics for expressing the ruleset in a way that satisfies the requirements in the previous section can be achieved by following a similar meta-modelling approach to that described in Section 4.1. This process dictates that the correct level of abstraction must be chosen, and then from that level of abstraction, the relevant entities selected. The level of abstraction is selected by analysing the key entities that exist in the domain being modelled, in this case expressing rules for the MobJeX system.

To determine abstraction and perspective, the use cases described in the previous section were analysed and the primary entities captured. These entities are shown along with the implied constraints in Figure 12. The *Ruleset* represents the top level collection of rules and constraints. A *Rule* represents a rule as described in the Literature Review in Section 2.3, which is composed of a set of boolean *Condition* clauses and a *Action* to take if all the stated conditions evaluate to true. A *Constraint*

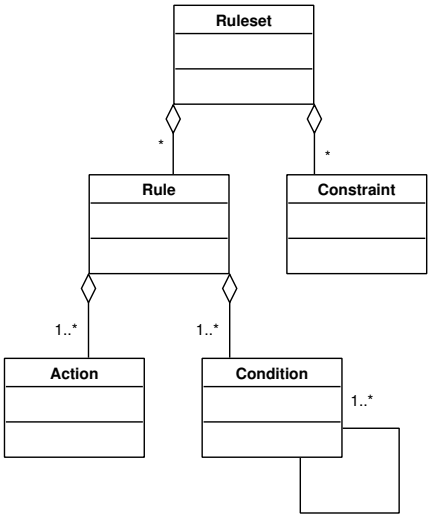

Figure 12: MobJeX Rule Entities

represents conditions that apply to all rules, or in simpler terms, a set of conditions which can't be made false.

Rules represent the majority of the use cases required by the MobJeX framework and model. Rules are used to describe any situation which fits the pattern of detecting a set of conditions in a *ModelEntity* and taking action to correct or adapt the system to those conditions. Detecting sets of conditions (often termed a 'scenario') is achieved by filtering the entities in the model through the closures (Bawden and Rees, 1988) that each condition provides and then processing the action component for entities that match. The closure conditions can be any combination of boolean conditions, including nested boolean expressions.

Constraints describe conditions in the model that must not be violated. Constraints differ from rules in that where a rule dictates actions that should occur when a set of conditions is true, a constraint dictates a set of conditions that must always hold true. A constraint is made up of a semantic description of the conditions that must be maintained, for example a group of objects which must always be local to each other.

Both conditions and constraints reference metrics provided by the model. Metrics provide a standard means of measuring aspects of the operating environment of the mobile application, as well as quantitative information about the application's functional characteristics (Rossi and Fernandez, 2003). The metrics that are collected and made available can be divided into two categories, resource metrics and code metrics (Fenton and Pfleeger, 1996). Resource metrics provide information about the operating environment, generally on a per-system basis. Examples of resource metrics would be available memory, CPU utilization, network interface load, etc. Code metrics are collected on a per-object basis and provide information about the objects themselves, such as the object's size, method count, frequency of method calls, times moved, etc.

The model collects metrics from the MobJeX mobility subsystem in two places: the service, which provides per-host resource metrics and from the object proxies, which provide code metrics about the object which they proxy. Metrics are collected by polling the MobJeX framework at fixed intervals. Several optimisations are possible for this process, such as pushing the metrics from the MobJeX sub-

system rather than pulling them from the model. This technique would allow for individual services and mobjects to decide when to send a bundle of metrics by caching them until a metric reaches a particular threshold. This approach allows for remote calls to be minimized at the cost of reducing the freshness (Bouzeghoub, 2004) of the metrics.

With the mechanisms for collecting and inquiring about metrics established, it becomes possible for the rule engine to "query" the model, selecting entities with matching metrics. This satisfies Usecase 1 and Usecase 2. Providing a mechanism for querying the metadata in the model allows for the specification of the concrete actions and constraints described in Usecase 3. These specific actions, *move*, *log*, *retry* and the *co-locality* constraint are described in detail in the following sections.

Actions

The *Move* action causes a mobject to move from its present location to either a specific named location, or a location that meets certain criteria, expressed as a list of metrics.

The *Log* action causes a debug or warning message to be recorded in a log file or sent via email to a user. This capability enables the iterative development of rules, as initially actions can be expressed as simple log messages to demonstrate that the constraints are working as expected, but when the rule is verified to work as expected, the actual move action can be substituted in.

The *Retry* action is used to overcome transient network errors as described in Section 4.5.1 by providing strategies for dealing with failing commands. *Retry* can either cause the failed command to be re-tried until it succeeds, or it can be given a fixed number of retries. In the case of the latter, *Retry* can be given an action to perform if the number of retries is met and the command still fails. This allows for an alternative strategy to be attempted or for an error message to be logged.

Constraints

Constraints can either be maintained passively or actively (Barbar-Mill and Garcia-Molina, 1994). Maintaining constraints passively is where the rule engine prevents rules from violating constraints, thereby maintaining the constraint. Maintaining rules actively is where the rule engine maintains constraints by triggering actions, rather than by preventing actions from occurring. Active constraint maintenance has the benefit of correcting situations where constraints are unavoidably violated, such as if the initial state of the model violates a constraint.

The *Co-Locality* constraint specifies that objects must be located on the same node. This provides a means for ensuring that objects with a high level of inter-object coupling keep remote calls to a minimum. As discussed in the literature review in Section 2.1, an alternative approach to this problem is to analyse the runtime structure of the objects and automatically determine which objects require this constraint, but this is beyond the scope of this thesis and represents a future area of work.

5.3 Syntax

XML was selected as the markup language for the rule formats due to its portability and expressiveness. The text-based nature of XML provides a more human-readable format than a more compact binary alternative. Additionally, the prevalence of tools for manipulating and editing XML provides users with a variety of methods for maintaining and manipulating rulesets. Libraries that support XML are available for parsing, searching and transformation. These features combine to make XML a choice that allows for an expressive rule syntax to be rapidly prototyped and for the result to be readable by users of the MobJeX framework. Future work may include the development of a custom

non-XML syntax to reduce the verbosity of an XML format and provide customisations specific to the MobJeX framework.

The specific rule format was created from the semantic entities described in the previous section. Actual syntax was derived from the entities by following three core principles of XML design (Ogbuji, 2004) which are summarized below.

The first principle is the *Principle of Core Content*. This principle dictates that information that is critical to the model conveyed by the XML document should be represented as elements. By contrast, information that is peripheral to the core content should be represented as attributes.

The second principle is the *Principle of Structured Information*, which states that if the information conveyed in the document exists in a structured form or has the potential to become structured in the future, express the major entities as elements. By contrast, if the document is composed of simple, atomic tokens of information represent them as attributes.

The final principle is the *Principle of element/attribute binding*. This states that attributes are strongly coupled to the element that they exist within. Due to this coupling, if a schema requires other elements to access or modify information in another elements attribute, the attribute should exist in an element of its own. Ogbuji succinctly summarizes this as "an attribute should only specify information about the element in which it is declared."

These principles were used to select which entities should become XML elements and which entities should be represented as attributes. The actual XML syntax is described in the context of a number of scenarios, along with relevant fragments of the rule format.

Scenario 1: A host is running low on memory

A simple distributed application is made up of three coarse components, a GUI, a data model and a logical controller. The application uses a classical Model-View-Controller approach to separate these components, connecting the GUI to the model via the controller. The model is large, so it requires as much free memory as possible, so when the host that the model is running on runs low on memory, the model is moved to another location with more free memory

```
1  <?xml version="1.0"?>
2  <ruleset xmlns="http://www.cs.rmit.edu.au/mobjex/ruleset/20040624">
3     <rule>
4        <match identifier="mobject">mobjex.model.MobjectModelEntity</match>
5        <match identifier="tm">mobjex.model.TransportManagerModelEntity</match>
6        <condition>mobject.metric("/system/memory/total_free_percent") &lt; 20</condition>
7        <condition>tm.metric("/system/memory/total_free_percent") &gt; 60</condition>
8        <move>
9           <source identifier="mobject" />
10          <destination identifier="tm" />
11       </move>
12    </rule>
13 </ruleset>
```

Figure 13: Moving a mobject to a runtime with more memory

Figure 13 shows a rule that matches a mobject and a transport manager. These entities are aliased to the short names (for later reference) 'mobject' and 'tm' (line 4 and 5). Two conditions are then presented, the first being that the mobject matched must be running on a host with less than 20% free memory (line 6) and the second being that the transport manager must be running on a host with

greater than 60% free memory (line 7). These conditions illustrate how it is possible to match multiple objects with a range of conditions.

The *move* element (line 8) indicates the action to perform when the above conditions are matched. Multiple actions are permissible at this level. The *move* element in the rule above moves the matched mobject to the matched transport manager. This creates a simple round-robin load balancing as the rule engine iteratively applies this rule to every over-burdened mobject in the model, moving it to a host with more free resources. The limitation of this rule is that it fails to take into account the quantity of free memory in the current host and the target host. For instance, the mobject might be moving from a host with 20% of 4Gb of memory free to a host with 60% of 128Mb free. This could be addressed by adding in a condition that ensures that the quantity of free memory in the destination host is much greater than that of the current host.

In the condition elements (line 6 and 7) a simple java-like syntax is used. This syntax uses the *object.method()* style that is employed in C++, Java and a number of scripting languages such as PHP. This syntax allows for basic boolean logic operators such as `or`, `and` and `not` as well as arithmetic operators such as <, >, == and `!=`. Characters which are reserved in XML such as <, > must be represented in XML entities (`<` and `>` respectively). The use of a simple programmatic syntax rather than expressing conditions entirely in XML allows for better readability due to the reduction in bulk and provides a compromise between flexible syntax and readability. This trade-off is made in several XML-based technologies such as XPath (Clark and DeRose, 1999) and XSLT (Clark, 1999).

The condition blocks reference the metric `/system/memory/total_free_percent`. This is one of the metrics made available to the rule engine via the model. The initial implementation provides some simple metrics such as free memory, CPU utilization and network utilization. The specification of which metrics are required by the MobJeX system and the implementation of all of these is an area for future work.

Scenario 2: A network has transient errors

An application is running on a network with a high degree of packet-loss. Commands often fail due to network-level errors.

```
1  <ruleset xmlns="http://www.cs.rmit.edu.au/mobjex/ruleset/20040624">
2     <rule>
3        <match identifier="command">mobjex.model.commands.Command</match>
4        <condition>command.state == FAILED</condition>
5        <retry times="5">
6           <command identifier="command" />
7           <on-fail>
8              <log destination="stdout">A command failed to execute 5 times</log>
9           </on-fail>
10       </retry>
11    </rule>
12 </ruleset>
```

Figure 14: Retrying a Command

Figure 14 shows a rule that matches a command (line 3) that has a failed (line 4). The action is to retry the matched command (line 6) five times (line 5) and if its still failing (line 7), log a message to the console (line 8). The model is used to track the number of failed commands.

Scenario 3: Objects need to be located on the same host

A distributed application has 3 mobjects which make a large number of calls to each other. To minimize call times between these highly coupled objects they must be located on the same host.

```
1  <ruleset xmlns="http://www.cs.rmit.edu.au/mobjex/ruleset/20040624">
2     <constraints>
3        <co-locality>
4           <match uri="mobjex://localhost:10000/*/test-app/mobject1" />
5           <match uri="mobjex://localhost:10001/24/test-app/mobject2" />
6           <match uri="mobjex://localhost:10002/24/test-app/mobject3" />
7        </co-locality>
8     </constraints>
9  </ruleset>
```

Figure 15: Ensuring Objects Remain Local

Figure 15 shows a constraint (line 2) that ensures that mobject1, mobject2 and mobject3 (line 4-6) are located on the same host (line 3). The *co-locality* element contains three *match* elements which reference mobjects via URI, as described in Section 4.2. URIs allow for objects to be specified by location, rather than by metrics or attributes as in the previous scenarios. This illustrates the interaction between the rule engine and the model in answer to Research Question 4, as the model provides a means for indexing entities based on URI and searching them when requested by the rule engine. The first *match* element shows the use of a wildcard character (line 5) which expands to match similar URIs with any runtime in the position of the wildcard character.

5.4 Implementation

A specification for Java-based Rule Evaluation Engines is at the time of writing undergoing the community process program. Java Specification Request (JSR) 94[5] describes an API for "acquiring and using a rule engine", where a Rule Engine is defined as an interpreter that when given a data set of objects and a set of rules will traverse the objects and apply the rules, as defined by the semantics of the particular rule language. JSR-94 states that the specification of the particular rule language are outside of the scope of the spec, as are the semantics of rule evaluation.

The API proposed by JSR-94 has a number of available implementations, both open-source and proprietary. As this API is the currently accepted industry standard for integrating different rule engines implementations, it provides an excellent point to adapt an existing rule engine framework to the proposed rule format's needs.

Two implementations were selected for review, Jess[6] and Drools[7]. The two engines evaluated support a database of Java-based objects that form the 'facts' (Milliken et al., 1988) that the conditions in rules evaluate. Additionally, all of the engines support forward-chaining using the RETE algorithm (Forgy, 1982) which allows for related facts to be connected in a chain in memory such that performance can be independent of the total size of the ruleset. This feature allows rulesets to trade increased memory-usage characteristics for the ability to scale evaluation performance.

Jess is a rule engine and scripting environment which extends on the syntax of the widely-used

[5] http://www.jcp.org/aboutJava/communityprocess/review/jsr094/

[6] http://herzberg.ca.sandia.gov/jess/index.shtml, as of 13/09/2004

[7] http://drools.org/, as of 13/09/2004

CLIPS[8] system, which is an environment for the construction of expert systems. Jess provides a comprehensive API for rule-based applications, however as it is historically derived from a language for creating expert systems, the supporting applications and examples are largely from that domain. Jess is licensed under a proprietary licence that prohibits source code redistribution and modification, so whilst Jess may be free for academic institutions, it provides a less than optimal platform for modifications. Jess provides excellent Java integration and an interface that is compliant with JSR-94. However the rule syntax, which resembles LISP, is verbose and non-XML based, meaning that translation from the proposed rule syntax to the Jess syntax would be non-trivial.

Drools is a rule evaluation engine that focuses on the evaluation of 'business rules' using forward-chaining. Like Jess, Drools provides an interpreter for a declarative rule and fact based reasoning language. Unlike Jess, Drools was not implemented to fulfil a need for CLIPS-compatible reasoning for expert systems, rather it was constructed from scratch to meet the demands of many applications that require deductive rule-based reasoning. Drools provides a JSR-94 compatible interface and offers an XML-based rules language that uses 'pluggable semantic modules' to allow conditions and actions to be implemented in scripting languages. This prevents the need for extensions to the rule engine itself, as actions can be implemented in the underlying Java system, in this case the model as described in Section 4. Drools is available under an open-source licence.

```
1  <?xml version="1.0"?>
2  <rule-set name="HelloWorld"
3        xmlns="http://drools.org/rules"
4        xmlns:java="http://drools.org/semantics/java">
5
6     <rule name="load balance low-memory services">
7        <parameter identifier="entity">
8           <java:class>mobjex.model.MobjectModelEntity</java:class>
9        </parameter>
10       <parameter identifier="dest">
11          <java:class>mobjex.model.TransportManagerModelEntity</java:class>
12       </parameter>
13       <java:condition>entity.getMetric("free_memory_percent") &gt; 20</java:condition>
14       <java:condition>dest.getMetric("free_memory_percent") &lt; 60</java:condition>
15       <java:consequence>
16          RuntimeURI runtimeURI = ((TransportManagerURI) dest.getURI()).asRuntimeURI();
17
18          controller.handleCommand(MoveCommand.create(CommandState.PENDING, entity.getURI(),
19             runtimeURI));
20
21          entity.updateMetrics();
22          dest.updateMetrics();
23
24          drools.modifyObject( dest );
25          drools.modifyObject( entity );
26       </java:consequence>
27    </rule>
28
29 </rule-set>
```

Figure 16: Drools Syntax For Moving Objects

Of the two candidates reviewed, Drools was selected to provide the underlying functionality of the implementation. The syntax for Drools rulesets is represented in XML, but makes heavy use of embedded Java fragments for specifying rule conditions and the action taken when the rule is matched. Figure 16 shows the syntax required for moving a mobject from a system with low memory. As the

[8] http://www.ghg.net/clips, as of 13/09/2004

example makes heavy use of Java syntax, it hinders its readability for non-programmers. The proposed ruleset syntax that performs the same function as the above figure is shown in Figure 13 in the first scenario in the Section 5.3. The proposed syntax is more domain-focused, in that the vocabulary used is that of the MobJeX framework, rather than a Java-based rule system. As the Drool syntax differ substantially from the proposed format, an XML transformation style sheet (XSLT) is used to convert between the two different syntaxes. As the Drools format is represented in XML, using a transform between the two formats allows for the Drools syntax to be isolated from the rest of the system, which provides the possibility of using an alternate rule engine if a more appropriate candidate becomes available.

5.5 Testing and Results

A number of tests were conducted to evaluate the integration of the model with the ruleset, in reference to Research Question 4. Tests were conducted against a model that was modified to allow metrics to be manually set for a particular entity. Tests were conducted in the same system environment described in Section 4.3.2.

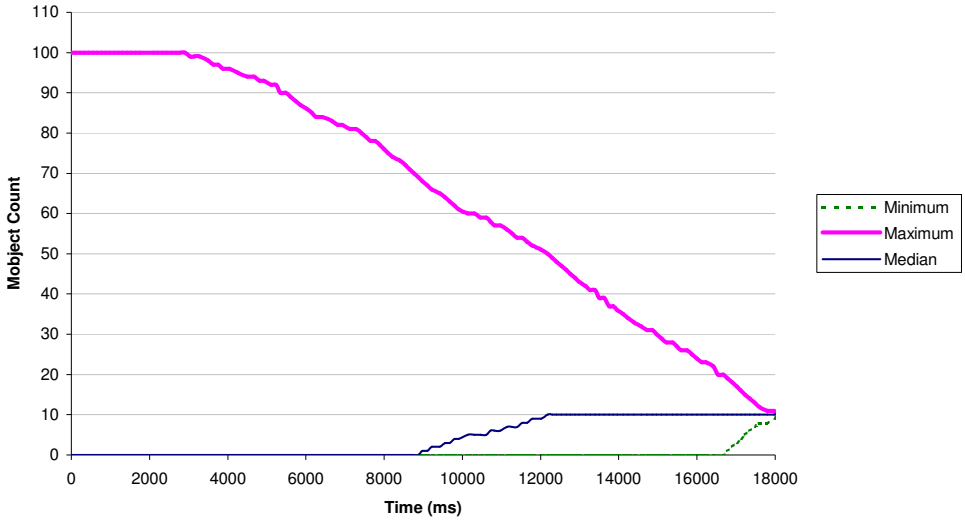

Figure 17: Load Balancing the Model

The first test conducted was a system test to ensure that the rule engine and the model operate as expected and correctly interact with the MobJeX system. A ruleset with a single rule was created. The rule matched a transport manager with more mobjects than the current average mobject count. A second transport manager was also matched that contained less than the average mobject count. When this rule was matched, the mobject was moved to the more sparsely populated transport manager. When applied to the model, the net effect of this rule is to distribute objects evenly across all of the transport managers.

A model was created with a single service. Within the service, 10 runtimes, each with a transport manager, were created. The first transport manager was populated with 100 mobjects, but the other 9 transport managers were left empty. Once the system was initialised, the ruleset was executed until every transport manager had an equal number of mobjects.

As figure 17 shows, the system balanced objects evenly across the 10 runtimes, resulting in the maximum, minimum and median value all gradiating towards the average point of 10 at the 18 second

mark. Whilst this test used a simple metric, the result demonstrates the rule engines ability to apply rules to the model. This technique could be expanded to use a metric such as 'Free Memory' or 'Network Utilization' and the result would be of greater utility in a distributed environment. Far more complicated rulesets are possible, but the full investigation of these possibilities and the performance implications are beyond the scope of this thesis.

In addition to collecting statistics pertaining to each transport manager in the model, a counter was added to the system controller that incremented for every MoveCommand received. This allowed for the efficiency of the load balancing algorithm used to be measured. As the load balancing algorithm used in the rule set rotates mobjects from crowded hosts to more sparse hosts in a round robin fashion, the amount of move commands required for any quantity of runtimes and mobjects can be calculated with the formula $mobjects - (mobjects/runtimes) = minimummovements$. For the previous experiment, this equation held true, as load balancing 100 mobjects to 10 runtimes required 90 move commands.

6 Conclusion

Mobile applications represent a relatively new area of distributed applications and dealing with the problems mobility brings is a distinct challenge. Mobile object frameworks provide a means of transparently distributing existing applications amongst a network of mobile nodes, and a mechanism for transparently relocating objects within this network. A variety of attempts have been made to build a solution that allows for flexible runtime reconfiguration and adaptation of the ever-changing network of objects that these frameworks present. At the time of writing, none of these attempts have provided a solution that allows for mobile applications to respond and adapt to the changing conditions they are confronted with in a manner that is transparent to the application developer. This paper has presented a solution to this problem in the form of two distinct, but complimentary components that build upon the MobJeX object mobility framework.

A model was implemented and tested which represents the state of an entire distributed application. This model provides the mobile objects in the system with unique identity in the form of URIs, and tracks the movements of these objects within the system. The model allows for fast, efficient lookups on the quantitative metrics stored about each object and provides a generic interface for causing change in the framework via asynchronous commands. This addition allows for application developers to develop applications which communicate with the distributed application as a whole, as opposed to the previous solutions which require that the application being distributed be responsible for the monitoring and maintenance of the remote objects. A further area of investigation in this area is the addition of security and authentication features for the remote communications methods used by the model

A ruleset for describing conditions within the model and actions to take to correct or improve the detected conditions was specified, along with a rule engine to interpret these rules and provide the ruleset with access to the rich range of metadata and metrics provided by the model. The rule engine provides users of mobile applications with a means for specifying deployment rules in a declarative, dynamic way. Rules are processed in real-time as the mobile application runs, allowing for rules to be specified that optimise the mobile application by moving objects to the most appropriate locations. These rules are specified in a simple, human-readable format that allows for mobile applications to be adapted at runtime without any modifications to the code of the original application. A further area of research in this area would be the investigation of more complex rulesets, the performance implications of each rule and a means of generating rulesets automatically based on dynamic analysis

of the application.

The model and rule engine were integrated with the MobJeX framework to provide a solution which allows for mobile applications to be transparently distributed over ad hoc networks of heterogenous devices. The contributed model and rule engine allow for the MobJeX framework to adapt to changes in the environment in real-time by moving mobjects to appropriate devices.

References

Adi, A. and Etzion, O. (2004). Amit - the situation manager. *The VLDB Journal*, 13(2):177–203.

Barbar-Mill, D. and Garcia-Molina, H. (1994). The demarcation protocol: a technique for maintaining constraints in distributed database systems. *The VLDB Journal*, 3(3):325–353.

Baude, F., Caromel, D., Huet, F., and Vayssière, J. (2000). Communicating mobile active objects in java. In Marian Bubak, Hamideh Afsarmanesh, R. W. and Hetrzberger, B., editors, *Proceedings of HPCN Europe 2000*, volume 1823 of *LNCS*, pages 633–643. Springer.

Bawden, A. and Rees, J. (1988). Syntactic closures. In *Proceedings of the 1988 ACM conference on LISP and functional programming*, pages 86–95. ACM Press.

Beeri, C. and Milo, T. (1991). A model for active object oriented databases. In *Proceedings of the 17th International Conference on Very Large Data Bases*, pages 337–349. Morgan Kaufmann Publishers Inc.

Ben-Shaul, I., Gazit, H., Gidron, Y., Holder, O., and Lavva, B. (1999). Fargo: a system for mobile component-based application development. In *Proceedings of the 21st international conference on Software engineering*, pages 658–659. IEEE Computer Society Press.

Benjamin, P., Erraguntla, M., Delen, D., and Mayer, R. (1998). Simulation modeling at multiple levels of abstraction. In *Proceedings of the 30th conference on Winter simulation*, pages 391–398. IEEE Computer Society Press.

Berners-Lee, T., Fielding, R., and Masinter, L. (1998). Uniform resource identifiers (uri): Generic syntax.

Bouzeghoub, M. (2004). A framework for analysis of data freshness. In *Proceedings of the 2004 international workshop on Information quality in informational systems*, pages 59–67. ACM Press.

Chan, A. T., IEEE, and Chung, S.-N. (2003). Mobipads: A reflective middleware for context-aware mobile computing. *IEEE Transactions on Software Engineering*, 29(12).

Clark, J. (1999). Xsl transformations (xslt) version 1.0. Available at http://www.w3.org/TR/xslt.

Clark, J. and DeRose, S. (1999). Xml path language (xpath) version 1.0. Available at http://www.w3.org/TR/xpath.

Cornelio, A. and Navathe, S. B. (1993). Applying active database models for simulation. In *Proceedings of the 25th conference on Winter simulation*, pages 535–543. ACM Press.

Dewan, H., Stolfo, S., Hernandez, M., and Hwang, J. (1994). Predictive dynamic load balancing of parallel and distributed rule and query processing. In *Proceedings of the 1994 ACM SIGMOD international conference on Management of data*, pages 277–288. ACM Press.

Elnozahy, E. N. M., Alvisi, L., Wang, Y.-M., and Johnson, D. B. (2002). A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408.

Emmerich, W. (2002). Distributed component technologies and their software engineering implications. In *Proceedings of the 24th international conference on Software engineering*, pages 537–546. ACM Press.

Eugster, P. T., Felber, P. A., Guerraoui, R., and Kermarrec, A.-M. (2003). The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131.

Fenton, N. and Pfleeger, S. (1996). *Software Metrics: A Rigorous and Practical Approach, Second ed.* London: International Thompson Computer Press.

Forgy, C. (1982). Rete: A fast algorithm for the many pattern/many object pattern match problem. In *Artificial Intelligence*, volume 19, pages 17–37.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (2003). *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Ghosh, B. (1994). Dynamic load balancing in parallel and distributed networks by random matchings (extended abstract). In *Proceedings of the sixth annual ACM symposium on Parallel algorithms and architectures*, pages 226–235. ACM Press.

Gotts, N. M., Polhill, J. G., and Law, A. N. R. (2003). Agent-based simulation in the study of social dilemmas. *Artif. Intell. Rev.*, 19(1):3–92.

Halevi, S. and Krawczyk, H. (1998). Public-key cryptography and password protocols. In *Proceedings of the 5th ACM conference on Computer and communications security*, pages 122–131. ACM Press.

Haumacher., M. (2003). JavaParty 1.07g. Available on: http://wwwipd.ira.uka.de/JavaParty. release November, 3rd 2003.

Hawick, K. A. and James, H. A. (2003). Middleware for context sensitive mobile applications. In *Proceedings of the Australasian information security workshop conference on ACSW frontiers 2003*, pages 133–141. Australian Computer Society, Inc.

Hong, J. and Bauer, M. (1993). A generic management framework for distributed applications. In *Proceedings of the IEEE First International Workshop on Systems Management*, pages 63–71.

Jing, J., Helal, A. S., and Elmagarmid, A. (1999). Client-server computing in mobile environments. *ACM Comput. Surv.*, 31(2):117–157.

Kappel, G. and Retschitzegger, W. (1998). The trigs active object-oriented database system an overview. *SIGMOD Rec.*, 27(3):36–41.

Kim, T. W., Kim, E. H., Kim, J. K., and Kim, T. Y. (1995). A leader election algorithm in a distributed computing system. In *Proceedings of the Fifth IEEE Computer Society Workshop on Future Trends of*, pages 481–485.

Kon, F., Costa, F., Blair, G., and Campbell, R. H. (2002). The case for reflective middleware. *Commun. ACM*, 45(6):33–38.

Law, A. M. and Kelton, W. D. (1997). *Simulation Modeling and Analysis*. McGraw-Hill Higher Education.

Malamos, A. G., Malamas, E. N., Varvarigou, T. A., and Ahuja, S. R. (2002). A model for availability of quality of service in distributed multimedia systems. *Multimedia Tools Appl.*, 16(3):207–230.

Milliken, K. R., Finkel, A. J., Klein, D. A., and Waite, N. B. (1988). Adding rule-based techniques to procedural languages. In *Proceedings of the first international conference on Industrial and engineering applications of artificial intelligence and expert systems*, pages 185–194. ACM Press.

Moyer, S. and Umar, A. (2001). The impact of network convergence on telecommunications software. *IEEE Comunications*.

Nassersharif, B., Peery, J. S., and DeHart, M. D. (1988). Highly vectorized algorithm for transient simulation of space reactor systems. In *Proceedings of the 1988 ACM/IEEE conference on Supercomputing*, pages 393–399. IEEE Computer Society Press.

Oehler, M. and Glenn, R. (1997). Hmac-md5 ip authentication with replay prevention.

Ogbuji, U. (2004). Principles of xml design: When to use elements versus attributes. Available at http://www-106.ibm.com/developerworks/xml/library/x-eleatt.html. as viewed on September, 3rd 2004.

O'Sullivan, D. (1997). Telecommunication applications of corba technologies. *Distributed Objects - Technology and Application*, 22(5):3–4.

Palaniswamy, A. C. and Wilsey, P. A. (1993). An analytical comparison of periodic checkpointing and incremental state saving. In *Proceedings of the seventh workshop on Parallel and distributed simulation*, pages 127–134. ACM Press.

Philippsen, M., Haumacher, B., and Nester, C. (2000). More efficient serialization and rmi for java. *Concurrency: Practice and Experience 12*, pages 495–518.

Randell, L. G., Holst, L. G., and Bolmsj, G. S. (1999). Incremental system development of large discrete-event simulation models. In *Proceedings of the 31st conference on Winter simulation*, pages 561–568. ACM Press.

Rossi, P. and Fernandez, G. (2003). Definition and validation of design metrics for distributed applications. In *Ninth International Software Metrics Symposium, 2003*.

Ryan, C. and Westhorpe, C. (2004). Application adaptation through transparent and portable object mobility in Java. In *Proceedings of 2004 International Symposium on Distributed Objects and Applications*.

Schmidt, D. C. (2002). Middleware for real-time and embedded systems. *Commun. ACM*, 45(6):43–48.

Schulz, M., Reck, F., Bartelheimer, W., and Ertl, T. (1999). Interactive visualization of fluid dynamics simulations in locally refined cartesian grids (case study). In *Proceedings of the conference on Visualization '99*, pages 413–416. IEEE Computer Society Press.

Shapiro, M. (1986). Structure and encapsulation in distributed systems: the Proxy Principle. In *Proc. 6th Intl. Conf. on Distributed Computing Systems*, pages 198–204, Cambridge, Mass. (USA). IEEE.

Spiegel, A. (2000). Automatic distribution in pangaea. *In Proc. Workshop on Communications-Based Systems (CBS 2000)*.

Sprinkle, J. (2004). Model-integrated computing. 23(1):28–30.

Terry, S. and Shawn, T. (2002). *Enterprise JMS Programming*. John Wiley & Sons, Inc.

Waldo, J. (1998). Remote procedure calls and java remote method invocation. *Distributed Objects - Technology and Application*, 6(3):5–7.

Watson, A. and Lovelock, J. (1983). Biological homeostasis of the global environment: the parable of daisyworld. *Tellus 35B*, pages 286–289.

Weinsberg, Y. and Ben-Shaul, I. (2002). A programming model and system support for disconnected-aware applications on resource-constrained devices. *Proceedings of the 24rd International Conference on Software Engineering, 2002 (ICSE 2002)*, pages 374–384.

Wu, J. and Popovi, Z. (2003). Realistic modeling of bird flight animations. *ACM Trans. Graph.*, 22(3):888–895.

Yoo, S., Kim, K., and Cha, S. (1999). A middleware implementation of active rules for odbms. In *Database Systems for Advanced Applications, 1999. Proceedings*.