Implementation and Evaluation of Scalability Techniques in the ECO Model

Mads Haahr
<Mads.Haahr@cs.tcd.ie>

Distributed Systems Laboratory
Department of Computer Science
University of Copenhagen
Denmark

Distributed Systems Group
Department of Computer Science
University of Dublin, Trinity College
Ireland

August 14, 1998

Abstract

Event-based communication is appropriate for many application domains, ranging from small, centralised applications such as GUIs to large, distributed applications such as telecommunications, network monitoring and virtual world support systems. Consequently, many different event models have been put forward, some designed for small-scale systems and others for large-scale systems. One such model is the ECO model which was designed to support virtual world applications in the Moonlight project. The ECO model was designed to be scalable by including filtering capabilities that were intended to decrease network traffic in a distributed implementation.

There have been two previous implementations of the ECO model, and one characteristic of both was that all code was linked into the application at compile time, regardless of whether it was used at runtime or not. This resulted in executables which were larger than strictly necessary, and consequently lower scalability for applications hosting many objects, most of which were used only on a few nodes. A better approach could be to link code on demand at runtime with the intention of decreasing the application footprint.

This thesis describes the design and implementation of a distributed version of the ECO model and the evaluation of filters and dynamic linking as means to achieve increased scalability. The evaluation is empirical and real data gathered from an actual event-based system is used. In addition to the design and evaluation chapters, a detailed review of four event models, including the ECO model, is provided, with particular emphasis on filtering and dynamic linking.

Contents

List of Figures

List of Tables

Preface

This thesis was written by Mads Haahr as part of the fulfilment of the Danish MSc degree (cand.scient). It describes the distributed implementation of an event-based programming model called ECO [Tea95] and evaluates two techniques for making the model and the implementation more scalable. The two techniques are event filtering and dynamic linking. The implementation is called SECO which stands for *Scalable ECO*.

The reader of this thesis is assumed to be familiar with distributed systems and object orientation and to have a basic knowledge of network technology. Also, some familiarity with the event-based programming paradigm, dynamic linking and the C++ programming language would be an advantage.

The author would like to thank Vinny Cahill of Trinity College Dublin and Eric Jul of the University of Copenhagen for their invaluable comments on this work and for their moral support and patience. Also, many thanks go to John Bates of the University of Cambridge for the invaluable data collected by their active badge system. Without this data, the conclusions presented in this thesis would have been much less objective.

Chapter 1

Introduction

1.1 Motivation

As long as anyone can remember, people have been fascinated by the creation and exploration of artificial worlds. Oral traditions, literature, painting, sculpture, film—any of the arts, really—can be viewed as the human mind reflecting the real world in an attempt to create artificial worlds of its own. To human creativity (and vanity) the idea of designing an entire world is compelling. It is therefore no wonder that virtual world software has quickly become popular and that the technology (in the form of *virtual reality*) has received much media attention, despite the fact that the applications are still at an early stage.

The emergence of relatively inexpensive rendering hardware has improved the graphical quality of single user worlds, and the increasing interconnectivity of computers has made it feasible to share virtual worlds between users. Still, however, current state of the art virtual world systems are limited to a relatively low number of users, in particular, in the light of the increasing popularity of the Internet. The problem of scalability—building very large virtual worlds with a very large number of users—is perhaps the greatest problem to be faced by virtual world researchers within the immediate future.[1]

Many different architectures for virtual world support have been put forward, and many of them include features to increase scalability. One such architecture is the VOID shell, developed as part of the Moonlight [Tea95] project. A central part of the VOID shell is an event-based programming model called ECO which includes event filters, called notify constraints. The purpose of notify constraints is to improve scalability by filtering events (dis-

[1] In section 1.5, we will look closer at what exactly defines scalability.

carding those that will not be missed) before they are transmitted across the network, and thereby decrease network traffic. Though the ECO model and its notify constraints have been implemented and used in the Moonlight project, no attempts have been made so far to actually evaluate notify constraints as a means to improve scalability.

1.2 This Thesis

This thesis describes the implementation of a distributed version of the ECO model and its notify constraints and attempts to evaluate them to see whether they offer any significant advantage in terms of saved network traffic. Also, the implementation includes another feature, dynamic linking, which is used to improve scalability by reducing size of the application executable. In this thesis, we generally use the term *footprint* to refer to the size of a program executable.

The approach to evaluating the scalability techniques is empirical rather than theoretical. A fully functional implementation of the event model is used in conjunction with real data, collected from an actual event-based system.

1.3 The Event Paradigm

One general characteristic of virtual world applications is that their communication pattern is often different from that of traditional client/server-based applications. When the client in a client/server system invokes a server procedure, communication is one-to-one.[2] Also, the client often (though not always) expects a reply from the server in the form of a return value. For these reasons, Remote Procedure Calls (RPCs) are typically implemented in a synchronous fashion, meaning that the client waits for the server's response.[3] This is depicted in the left-hand side of figure 1.1 on page 10. In the client/server model, only the client can initiate communication.

In virtual world applications, the client and server roles are replaced with the concept of peers. A virtual world is populated by a number of objects which can change state (e.g., move around) autonomously or as the result of user input. When an object changes state, it informs its peers (i.e., the other objects in the world) and in some models, such as ECO, this is done by

[2] If the server is in fact a cluster of servers acting as one, communication may really be one-to-many, but conceptually it is still one client invoking one service.

[3] It is possible to implement asynchronous RPC with return values, but it drastically increases the complexity of the programming model. Similar functionality without changing the model can be achieved by using synchronous RPC in conjunction with multi-threading.

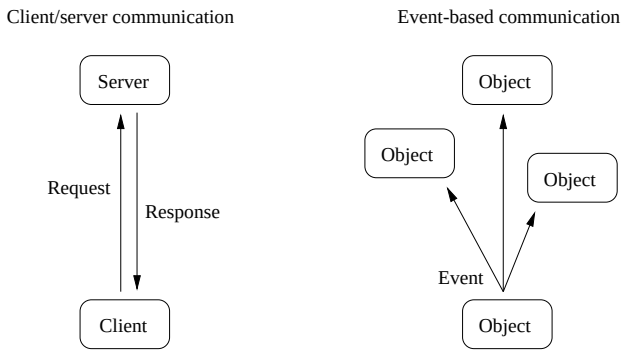

Figure 1.1: Client/server and Event-based Communication

raising a so-called event. Most often, there will be more than one receiver of such an event, and therefore communication is typically one-to-many. Also, there is no return value associated with an event. This is depicted in the right-hand side of figure 1.1. These characteristics mean that parties using event-based communication are typically more decoupled than those using an RPC model. Many (though not all) event models decouple parties even further by making event delivery asynchronous and in some models (such as ECO) the parties are also unaware of each others' identities. In the event paradigm, as opposed to the client/server model, all objects have the ability to initiate communcation (i.e., raise events) as well as accept communication requests (i.e., receive events). Propagating all events to all peers in the manner described obviously scales very poorly. Therefore, most event models allow objects to *register interest* in events. Registration lets an object specify exactly what events it is interested in by giving a set of criteria, and only events fulfilling these criteria are subsequently delivered to the object. Each event model has its own way of specifying what events are interesting, but most rely on event type and/or parameter matching.

The asynchronous one-to-many communications pattern is not specific to virtual worlds. It occurs in other domains, such as telecommunications systems and network monitoring and is also used on a smaller scale in graphical user interfaces such as X-Windows and Microsoft Windows. The ECO event model considered in this thesis was originally part of a virtual world toolkit, the VOID shell [Tea95], but is generic and can easily be applied to other application domains.

1.4 The ECO Event Model

As mentioned, the ECO model is a general-purpose event model. Applications using ECO support will be able to communicate asynchronously in a one-to-many fashion as outlined in section 1.3. The model specification falls in two parts: a set of concept definitions and a set of operations. This section provides a brief introduction by looking first at the former and then at the latter. A more detailed treatment of the ECO model can be found in chapter 3.

1.4.1 Concepts

The name ECO is short for *events, constraints,* and *objects* which are the three key concepts in the model. The *objects* are often referred to as *entities* to avoid confusion with programming language objects. Figure 1.2 shows the three ECO concepts in relation to each other. The basic idea is that entities use events to communicate and constraints to restrict their incoming communication. In the figure, Entity A on the left raises an event which is sent to Entity B on the right. On its way, the event is matched against the constraint which may or may not let the event through. The constraint is imposed by Entity B, and Entity A is not aware of the existence of the constraint.

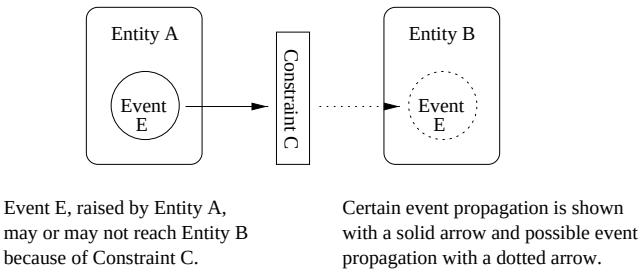

Figure 1.2: The Three ECO Concepts in Relation

An application using an ECO implementation typically has a number of entities communicating in this manner. Each entity can raise events as it pleases and create constraints to limit the number of events it receives. An ECO implementation must allow an application to create entities, constraints, and objects.

Additional Concepts

When discussing the ECO model in a virtual world context, we will use the term *user* to mean a person controlling one or more entities in the virtual

world. When discussing ECO in the context of distribution, the term *node* will be used to mean a physical machine hosting one or more entities. Nodes are assumed to be connected to a network.

1.4.2 Operations

There are three ECO operations: *subscribe, unsubscribe,* and *raise*. The first is used to register interest in events (as discussed briefly in section 1.3) and optionally insert a constraint as shown in figure 1.2. The second is used to cancel previously registered interest in events. The third is used to raise events, causing them to be propagated to other entities that have registered interest in them.

1.5 What is Scalability?

The term *scalability* has become something of a buzzword in the computer industry. There is no generally accepted scientific definition of what exactly scalability *is*, and people tend to rely on an intuitive understanding of the concept instead. Textbooks generally provide rather vague definitions and rely on examples to explain it. One of the more tangible definitions was made in connection with distributed garbage collection,

> *Scale is a relative concept that is hard to characterize precisely; rather we define scalability as a property related to an algorithm: it is scalable if its cost increases much slower than the number of spaces or of sites in the system.* [SPFA94]

Though somewhat specific to distributed garbage collection, the above definition makes the important observation that scalability is an algorithmic issue. To make the definition more general, *spaces* and *sites* should be interpreted according to the underlying domain. With regards to a virtual world model such as ECO, the parameters that can vary are,

- number of users
- number of entities (or objects)
- number of nodes (or machines)
- activity (communication)

Note that these parameters are in principle independent but in practice they are likely to increase together. For example, it is perfectly possible to have a very large and busy virtual world with many objects spread over many nodes but only inhabited by a single user—but it is not very likely. Also note that this list does not claim to be exhaustive for any virtual world model.

The implementation described in this thesis addresses the scalability issues mentioned above in several ways. In order to scale to a large number of users and entities, the system is *distributed:* As the numbers of users and entities increase, new nodes can be added to hold them. However, when world activity increases, communication between entities also increases and for entities residing on different nodes, communication must be performed over the network. Inter-node communication is more expensive than intra-node ditto, and if a world with many busy entities is spread over a large number of nodes the network can easily become a bottleneck. Hence, increasing the number of nodes may improve scalability in one way (to allow for many users and entities) but at the same time degrade it in another (for high levels of activity). The implementation described in this thesis addresses this problem by using *distribution* to scale the number of users and entities and *notify constraints* to minimise the amount of communication generated when scaling activity.

Though no formal definition exists, there seems to be a general agreement in the research community as to what scalability is, and we will not attempt to make a more precise definition than that presented above.

1.6 Project Objectives

As mentioned, the overall objective of this thesis is to evaluate two scalability techniques—filtering and dynamic linking—in the ECO model. This is a very abstract goal, and in order to make it more concrete, this section divides it into three more tangible subgoals called *Objectives A, B* and *C.* The three objectives were chosen such that each represents a logically coherent task which can be evaluated separately. The fulfillment of *Objective A* (a working implementation of the ECO model) is a prerequisite for the two other objectives but there are no other dependencies.

1.6.1 Objective A: Distribution

The ECO model itself does not specify any particular implementation approach for propagating events. Different solutions are possible, ranging from the strictly centralised (where one server is responsible for all event propagation) to partially centralised (where a cluster of servers cooperate) to fully

distributed (where no centralised server exists but where nodes exchange events directly). The original ECO implementation [Tea95] was centralised.

One objective of this project is to provide a fully distributed implementation of the ECO model. The objective is a proof of concept—a demonstration that such an implementation is indeed possible and usable. We do not, however, have a sufficiently large number of nodes available to evaluate the implementation in a large-scale configuration. Therefore, retaining maxiumum efficiency with a very large number of nodes is not a major implementation concern, but is discussed in the design and evaluation chapters.

1.6.2 Objective B: Dynamic Linking

In section 1.5 we mention that an application can be said to increase in scale if the number of entities it hosts increases. A related situation is if the number of entity types—or more generally, classes—increases. One problem with previous ECO implementations [Tea95] [O'C97] is that all class code is linked statically into the application at compile time. This means that the footprint of the program running on each node increases in size with every new type of event, constraint, or object. It also means that new code cannot be introduced at runtime. For some applications, the static approach means that code will be in memory[4] which may never be used. For such applications, it would be attractive to have code linked into the application on demand. For long-running applications, another benefit of dynamic linking would be the ability to change incrementally by replacing existing class code with new code during execution.

Our second objective is to provide support for dynamic linking of application code in our implementation of the ECO model. We want to determine how much extra complexity (in terms of code) is needed from the application to use this facility and to measure how large the gain is in terms of decreased application footprint. In order to limit the amount of work involved, the dynamic linking support will be limited to one type of code: notify constraints. However, a system design consideration will be that dynamic linking (if worthwhile) may later be extended to other types of code, such as events and objects.

1.6.3 Objective C: Measure Network Traffic Reduction

The last objective is one of investigation. As mentioned in section 1.1, the ECO model has filtering capabilities based on the use of notify constraints.

[4] Or on disk, if swapped out.

The primary motivation for introducing notify constraints in the model was to reduce network traffic by filtering events at their source. There is, however, a certain overhead involved in using filters, in particular with regards to subscriptions. One of our goals is to measure the number of network messages saved by the use of such filtering for a test application, and to judge whether notify constraints offer any significant reduction in network traffic.

1.7 Reaching the Objectives

The structure of this thesis—the choice of chapters and the order in which they appear—is to a large extent determined by the three objectives from section 1.6. This section motivates the choice of chapters based on the three objectives.

In order to fulfill *Objective A*, the work described in this thesis includes the construction of a distributed event service. This is a typical software engineering task, and we therefore describe the system in the traditional way by separating the description into analysis (or design) and implementation parts.

Objectives B and *C* require a series of experiments to be run on the system. These experiments, the results and the discussion/evaluation of the results are presented in the form of a single chapter.

The event paradigm is of extreme importance to this project, but the paradigm is not quite as well-known as, for example, the RPC model mentioned in section 1.3. It is therefore necessary to give the reader a good general understanding of the event concept, in particular with regards to filtering and dynamic linking. We attempt to do this with a fairly extensive state of the art review, which looks closely at three different event services, giving special attention to filters and dynamic linking. After the review, the ECO model itself—the event model used in this thesis—is described in detail in a separate chapter. The purpose of this approach is not to directly address any of the three objectives but to provide the reader with information about event systems in general, and the ECO model in particular, in order to thoroughly understand the subsequent treatment of the three objectives.

1.8 Document Structure

This section outlines the structure of this thesis. There are eight chapters in total, the order and contents of which are listed below.

Chapter 1: Introduction describes the motivation behind the project, introduces the central concepts and lists the objectives.

Chapter 2: State of the Art looks at related work in the area of event services and distributed virtual world support.

Chapter 3: The ECO Model looks at our event model from a theoretical viewpoint and takes an abstract look at the algorithms in a distributed ECO implementation. Also, we discuss the previous ECO implementations.

Chapter 4: Analysis analyses the problem domain by listing problems and proposing and discussing different solutions to each. It also sums up the design decisions and concludes with a system overview.

Chapter 5: Implementation describes the system implementation by giving an overview of the various components and the interaction between them.

Chapter 6: The Active Badge System describes an event-based system, a simulation of which is used to conduct experiments.

Chapter 7: Experiments and Evaluation describes and evaluates the experiments performed on the system.

Chapter 8: Conclusion concludes the project.

Chapter 2

State of the Art

Event and notification services are used in many different application domains ranging from small-scale systems, such as Graphical User Interfaces (GUIs), to telecommunications systems and network monitoring applications. In this chapter, we review the state of the art in general-purpose event services and describe three different approaches to expressing filters.[1] First, we review three event models, two of which are commercial and one of which is a research system. Section 2.1, discusses the event model included in JavaBeans (Sun's component architecture for Java), which is designed for small-scale applications. Section 2.2 reviews the CORBA event and notification services, which are designed to be extremely general-purpose and usable in virtually any domain. Section 2.3 dicusses the experimental event model developed at the University of Cambridge, which features an advanced concept called event composition. Section 2.4 then uses an example application to describe three different approaches (chosen on the basis of the three event models) to expressing filters. The discussion identifies and describes advantages and disadvantages of each approach. Section 2.5 summarises the chapter.

2.1 JavaBeans

Java is an object-oriented programming language, reminiscent of C++, which has become increasingly popular since it was launched by Sun in the mid 1990s. JavaBeans is a component model for Java also developed by Sun, and version 1.01 of the JavaBeans specification [Mic97] defines an event model. The model is designed with small centralised systems (e.g., window toolkits)

[1] For a recent and detailed review of a number of virtual world systems with event support see also [O'C97].

in mind but can be used in a distributed fashion by using the Java Remote Method Invocation (RMI) system. The JavaBeans event model was chosen amongst many comparable models (e.g., those found in X-Windows and MS Windows) as an example of a small-scale event model. This particular model was chosen primarily because of the increasing popularity of the Java language. The JavaBeans event model is based around two interfaces included in Sun's Java Development Kit (JDK) 1.1,

```
java.util.EventObject
java.util.EventListener
```

Also part of the model is a set of guidelines for method naming which applications should follow. By extending the two interfaces and implementing classes that support them in the manner described in [Mic97], Java applications can implement simple event-based communication.

2.1.1 Architecture

The model is based on the client-server paradigm. Event consumers (clients) are called *listeners* and event suppliers (servers) are called *sources*. A source announcing an event is said to *fire* it. In the simplest scenario, a listener registers interest in events fired by a particular source by invoking an `AddListener` method on that source. Subsequent occurrences of events from the source will cause a method to be invoked in the listener. The two parties are therefore very tightly coupled and there is no possibility for anonymity. Furthermore, events are always delivered to listeners synchronously, meaning that the source thread actually executes the listener's handler. Hence, a source with multiple listeners must, when it fires an event, deliver it to the listeners in sequence.[2]

The synchronous nature of JavaBeans event delivery means that the model has inherent performance penalties if used in a distributed environment. The widely accepted way of efficiently implementing distributed event delivery is to use network level multicast (such as IP multicast) to simultaneously deliver events to a number of receivers. Since the JavaBeans event model explicitly specifies that events should be delivered as a sequence of synchronous RMIs, each requiring a reply from the receiver,[3] network level multicast will be of little use in an implementation.

[2] In JavaBeans terminology, a source with multiple listeners is called a *multicast* source whereas one which only allows a single listener is called a *unicast* source. This is not to be confused with network level multicast and unicast.

[3] Even though there is no return value, a reply is still required to return the thread of control to the event source.

Filtering

There is no general support for filters in the JavaBeans event model, but the model includes a component—the event adaptor—where application-specific filters can be placed. An event adaptor is an object that can be inserted between the source and the listener to partially decouple communication between them as shown in figure 2.1. When using an event adaptor, a listener registering interest in events still invokes a method on the source directly, but instead of giving its own reference, it passes that of the event adaptor. When an event is later fired, the source delivers it to the event adaptor which in turn delivers it to the listener. This makes it possible to perform additional functions in the adaptor, such as event queueing or filtering. As opposed to other event models (e.g., the CORBA event service) the JavaBeans event adaptor approach is asymmetric; it only hides the listener from the source, not vice versa. Though the listener can retain some notion of anonymity using an event adaptor, it is still impossible for the source to be anonymous. Another consequence is that an event adaptor object must always be managed (i.e., created and destroyed) by the listener.

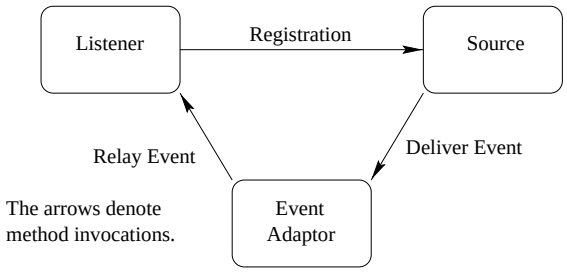

Figure 2.1: JavaBeans Event Model with an Event Adaptor

Dynamic Linking

The Java language has inherent support for linking of class code at runtime. This functionality is implemented by the `java.lang.ClassLoader` class described in [GJS96, p.558]. Runtime discovery of class interfaces is possible through the Java Core Reflection API. This means that new classes can be added to an application at runtime.

2.1.2 Summary

It is clear that the JavaBeans event model is built for event communication within a centralised or a small-scale distributed application. It has no in-

herent filtering support, cannot benefit from network level multicast, and because of the tight coupling between the source and listener requires that each party maintain detailed knowledge about the other. In its current form, it will not scale to be used in any virtual world, telecommunications, or network monitoring environment of substantial size. Support for dynamic linking is excellent, however.

2.2 CORBA Services

The Common Object Request Broker Architecture (CORBA) is a middleware architecture specificied by the Object Management Group (OMG). The architecture is based on the idea of using Object Request Brokers (ORBs) as a common way for different systems to perform remote procedure calls (RPCs). In addition to ORB functionality, the CORBA 2.0 specification [Gro95a] describes a number of general-purpose services, one of which is the CORBA Event Service. Applications using this service can communicate with events in addition to the normal RPCs provided by the bare ORB. Moreover, work is currently ongoing within the OMG to define a Notification Service to extend the event service with event filtering capabilities. The event models used in both services can be characterised as extremely general and quite complex. In the following two sections we review each of these models in turn.

2.2.1 Event Service Architecture

The CORBA Event Service [Gro95b] supports an event model where either the event supplier or consumer (collectively referred to as *clients*) can play the active part. In the former case, called the *push model*, the supplier raises events by *pushing* them to the consumers. In the latter case, called the *pull model*, the event consumer actively polls for events. Polling can be either blocking or non-blocking. The CORBA Event Service's *push model* corresponds to the paradigm most often found in other event models, e.g., the JavaBeans model described in section 2.1. In the simplest scenario, suppliers and consumers invoke each others' interface methods directly to obtain or deliver events. In this case, both parties require each others' object references and communication therefore cannot be anonymous. Anonymous communication, and other quality of service requirements, can be met by placing an object called an *event channel* between the two parties and letting each party interact with the channel instead.

Event Channels

Event channels in the CORBA Event Service act as intermediaries between suppliers and consumers. Since the channels are normal CORBA objects they can include any features the application designer wants to implement. Examples mentioned in [Gro95b] are quality of service features, such as event caching, persistence, anonymity, and filtering.

As shown in figure 2.2, channels have two *sides,* one towards the supplier and one towards the consumer. Each side has an `Admin` object associated with it which manages (e.g., creates and destroys) a number of proxy objects. There are two `Admin` interfaces, `ConsumerAdmin` and `SupplierAdmin`, each corresponding to one side of the event channel. Each proxy object is visible to one particular client on its side of the channel and acts as a stand-in for a client on the other side.[4] For example, a consumer talking to a supplier through an event channel communicates not with the actual supplier but with a stand-in object implementing the `ProxySupplier` interface.

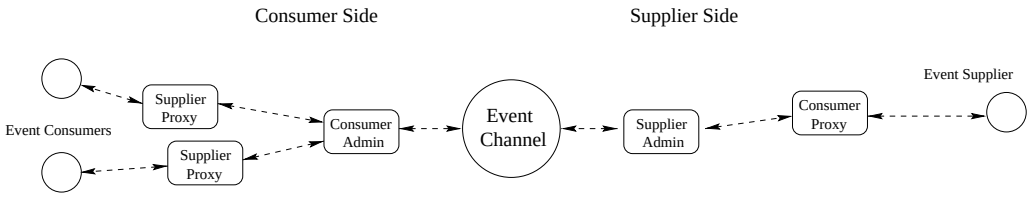

Figure 2.2: CORBA Event Service Overview

As mentioned, an implementor may choose to build additional functionality, beyond that of relaying events, into event channels. An event channel could for example save past events for purposes of error recovery or persistence. Past events could be cached for a while and retransmitted to consumers that have failed (or been unavailable for a period because of a network failure) during a recovery phase. Also, important events could be saved on persistent storage so they are still available and can be delivered to consumers in case the event channel itself is subject to temporary failure. Another possibility is to implement a filtering mechanism in event channels, but [Gro95b] sets no guidelines for this.[5]

[4] Hence, the same supplier can be represented by different supplier proxies on the consumer side, as shown in figure 2.2.

[5] In fact, the absence of a filtering mechanism was the primary reason for proposing the notification service described in section 2.2.2.

Dynamic Linking

In the CORBA event service, events can be declared as type `any`. This means that any object or data structure can be treated as an event and that new event types can be introduced into the application at runtime.[6] This is obviously a highly flexible feature but in order to use it the consumer needs to examine object type codes and learn about new event types at runtime. Typically, this would involve using the CORBA Interface Repository (IR) to examine the event interfaces. There is no support for dynamic linking in CORBA, since this is a language issue and CORBA is language-independent. Hence, whether dynamic linking is possible depends on the underlying operating system and implementation language.

2.2.2 Notification Service Architecture

In December 1996, the Object Management Group (OMG) issued a Request For Proposal (RFP) for a Notification Service [Gro96]. The idea was to extend the event service described in section 2.2.1 with filtering capabilities. At the time of writing, there are two such proposals: One [SDE+98] is submitted by a group of fourteen companies (IBM, Oracle and IONA among others) and the other [yDC98] by a group of two (one of them Hewlett-Packard). The two proposals solve the problem in similar ways.

Filtering

Both proposals extend the CORBA Event Service by allowing multiple `Admin` objects on each side of an event channel, as depicted in figure 2.3 on page 23. Filtering is done hierachically on each side. In both proposals, filters take the form of objects.[7] Each filter object holds a series of *constraints* which are text strings (in [SDE+98] the strings are further encapsulated in objects) containing filtering expressions. In addition, the filter object also holds an identifier of a grammar specifying the filter expression language. In [yDC98], filters can be assigned to `Admin` objects as well as to proxy objects, and in [SDE+98] also to the event channel itself. In both proposals, all filters (including those belonging to the channel and the admin objects) are evaluated

[6] Note that while both *objects* and *data structures* can be treated as events, only data structures can be passed by value. CORBA objects cannot be passed by value, only by reference. A distributed event-based application would probably pass events by value to avoid expensive remote invocations when the consumer accesses events. Therefore, such an application would typically implement events as data structures rather than objects.

[7] In [yDC98] filter objects implement the `MappingDiscriminator` interface, in [SDE+98] the `Filter` interface.

at the proxy level when an event is received.[8] Both proposals include a specification of a simple filter expression language which is interpreted at runtime. Other languages can be defined and coexist in the notification service. As opposed to most other models, filtering can be performed on the supplier as well as the consumer side, but each party is in control only of the filters on its own side.

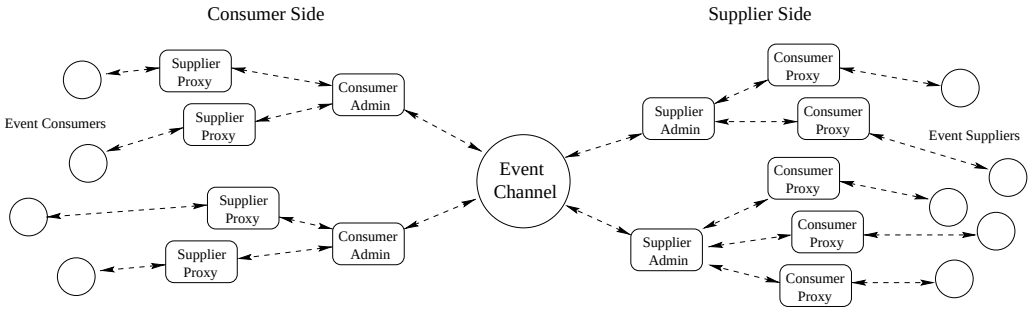

Figure 2.3: CORBA Notification Service Overview

As mentioned, a filter object holds a series of *constraints* which are string objects subject to interpretation in the context of a particular grammar. Constraints are also used for purposes other than filtering, namely to dynamically assign priorities to events and to set event lifetimes. To do this, consumers express rules that cause the event channel to change the characteristics of a particular event (such as lifetime and priority) if it satisfies the constraint. Of course, only the event actually destined for that consumer is changed; other instances of the same event destined for other consumers are not. Events are cloned to make this possible.

2.2.3 Summary

The CORBA Event and Notification Services have been designed to be usable in virtually any setting where event-based communication is required. The Notification Service, effectively a superset of the Event Service, constitutes an extremely general event model and allows for great flexibility in application design. The CORBA Interface Repository (IR) makes it possible to introduce new events at runtime and the possibilities for dynamic linking therefore depend only on the underlying implementation language. A good choice for this purpose could be Java which, as mentioned in section 2.1 has inherent support for dynamic linking. The filtering capabilities suggested in both Notification Service proposals will be another powerful feature, and

[8] [yDC98, p.3-18] and [SDE+98, p.37].

since filters are text strings which (as opposed to CORBA objects) can be transmitted by value, they have no class code which would need to be linked dynamically.[9] A feature supported neither by the Event Service nor the proposed Notification Services is the ability to *compose* events. In section 2.3 we shall look at a model that supports this concept. Apart from the lack of composition support, it is difficult to imagine a distributed event-based application that could not fit into the CORBA event model. However, the generality is paid for by an increase in complexity—understanding and using the model is difficult, and a correspondingly high development cost can be expected for applications using it.

2.3 Cambridge Event Model

The event model described in [BBHM95] and [Hay96] was developed at the University of Cambridge Computer Laboratory. It is architecturally much simpler than the CORBA model, but has a feature not present in the other, namely that of *event composition*. As opposed to the models reviewed in the previous sections, the Cambridge model is not a commercial product but the product of a research project.

2.3.1 Architecture

The Cambridge event model is based on the client-server paradigm. An *event service* in Cambridge terminology is a program which supplies events to clients, corresponding to the *push* model in the CORBA event service. The Cambridge model identifies three steps required for event communication: *event specification, registration* and *notification*.

The model includes an interface definition language (IDL[10]) which lets a service implementor *specify* the interfaces of a service. IDL code includes method definitions used for normal RPC but can also include event declarations, if the service in question can raise events. A preprocessor is used to translate the IDL code into client and server stub code and partial implementation of event classes. An example of an IDL definition for an active badge system given in [BBHM95] is,

```
Badge : INTERFACE =
  Seen: EVENTCLASS [ badge : BadgeId;
```

[9] The only code required to evaluate constraints is an interpreter for the standard constraint expression language. This interpreter is mandatory for any implementation of the notification service.

[10] Different from IDL specified by the OMG but with similar functionality.

```
                        sensor : SensorID ];
END.
```

This badge service raises an event of type *Seen* when it detects a badge. Identifiers of the badge and the sensor that saw it are included in the event as parameters.

Filtering

When a client *registers* interest in events, it can give a filter expression which specifies what events the client is interested in. Filters are text strings that take the form of event templates. In a template, the event parameters are replaced with expressions in a language similar to regular expressions but slightly extended. In order to match actual events against template strings, an application programmer uses a preprocessor to generate extra code for each event object. When invoked at runtime, this code produces a string template very much like the filter template but with the real event parameters instead of expressions. An event is matched against a filter by matching this template against the filter template. As we will see in section 2.4.3, there are certain limitations as to what can be expressed with event templates. Therefore, the filter templates have been augmented with *side expressions*. They are additional restrictions on the instantiated parameters and are expressed in a language reminiscent of C++ expressions. An example of a filter template given in [BBHM95] is,

```
Seen(P,42)
```

This template matches all events of type *Seen* originating from sensor 42. The template is really a regular expression which is evaluated by a finite state machine. The variable `P` acts as a wild card until it is instantiated by the state machine.

To raise an event, the service creates an event object and invokes an event service module. This module matches the event against registrations and relays the event to the clients. The event service module also handles all information related to client and server identities. Therefore neither party is aware of the identity of the other and anonymity is retained. Exactly what an event object looks like depends on the implementation language but as mentioned above, events can be converted to template form by preprocessor-generated code included in the event object. An example of such a template is,

```
Seen(12,42)
```

This event in string form means that badge 12 has been seen by sensor 42. This type of string is easy to match against filter templates.

Dynamic Linking

As for the CORBA event service, dynamic linking is not related to the event model itself but to the underlying platform. However, since filters are string templates, they do not require code for evaluation beyond that of the interpreter, and are therefore inherently mobile.

Composition

While the Cambridge model's way of expressing filters as event templates is elegant, its principal strength lies in its ability to compose these templates. Clients can specify that they are, for example, interested in the sequential occurrence of two particular events (but not in any of the events alone), any event from a set of events, or only in an event if another has not previously occurred.[11] Templates can be combined and very complex registrations can be performed. Finite state machines are used to process these registrations. An example of a composite event template could be,

```
// sensor 42 is in the reactor room
Seen(P,42);Seen(P,*)
```

This template would trace any person who has been sighted by sensor 42, reporting all subsequent sightings.

2.3.2 Summary

In many respects, the Cambridge event service is simple, elegant and easy to understand. It is less flexible than the CORBA service but also a lot less complex. As mentioned, its principal strength is that it has native support for composite events. Composition, as will be shown in section 2.4.3, is a powerful feature. It has yet to be discovered by industry, however, and so far it is not even mentioned in the event documents published by the OMG. Composition does not require dynamic linking facilities for filters, since filters are interpreted. With regards to dynamic linking of other classes of code, the possibilities depend on the application implementation language.

[11] The exact operations and their forms are given in [Hay96, pp.65–67].

2.4 Expressing Filters

The previous sections presented and discussed three different event models. Two of these, the CORBA and Cambridge models, included support for interpreted event filters and featured actual languages in which filters could be expressed. The third model included, JavaBeans, did not include explicit support for filters but an application could implement its own filters by using event adaptors. A central difference between this approach and the two others is that JavaBeans filters would have to be compiled (into Java byte code) rather than interpreted at runtime.[12]

In this section we take a closer look at how filters can be expressed. We discuss three possible approaches, one relying on a C++ style interpreted expression language similar to that of the two CORBA proposals, another being the regular expression language found in the Cambridge model and the last being a compiled approach, a variant of which could be used for example in the JavaBeans model. The first two are examples of interpreted and the latter of compiled filters. The purpose of the discussion is to present the strengths and weaknesses of the different approaches and contrast them with each other. To make the discussion easier to understand, we will first present an example—a simple tank game—that uses a filter, and then for each of the three approaches extend the example with actual filtering code. Section 2.4.1 describes the example, sections 2.4.2 to 2.4.4 the three approaches and section 2.4.5 concludes this section.

When we talk about filters on an abstract level, we assume them to be expressions that can be evaluated as a boolean value which indicates whether a particular event should pass through the filter or not. One or more event parameters typically occur in a filter expression. On a more concrete level, filters can be represented in many ways, for example as objects, functions or text strings. This section discusses filters at both the abstract and the concrete level.

2.4.1 Tank Game Example

Consider a tank game where players guide tanks around a two-dimensional world. Each tank entity receives input from the player controlling it and repeatedly calculates and announces its new geographical position within the world. This could be done with an event of a certain type, let us call it `NewPositionEvent`, which has three parameters, namely the identifier of the tank raising the event and the x and y coordinates of its new position.

[12] That the Java byte code itself is (almost always) subject to interpretation by a Java virtual machine is a different matter and does not concern us here.

Such a world could contain minefields with the property that any tank located within a minefield would stand a certain chance of being blown up each time it moved. This could be implemented with a minefield entitiy that subscribed to events of the type `NewPositionEvent` with a certain notify constraint. When a tank is blown up by a mine, the minefield would have to tell this to its surroundings, for example by raising a collision type event, but we will not go into details about that in this example. C++ code is given in the example, but any object-oriented language would do. Also, we use the ECO terminology defined in section 1.4 and assume that the following two operations exist,[13]

```
subscribe(Event, Handler, Constraint);
raise(Event);
```

In addition, each entity in the world is assumed to have a unique identifier.

Event Class

The event used by the tanks to announce new positions would be defined by the application (since it is specific to the tank game), likely in the form of a class derived from a base event class. It could look something like,

```
class Event { ... };  // Base class

class NewPositionEvent : public Event {
  NewPositionEvent(entity_id, int new_x, int new_y);  // Constructor
};
```

Tank Entity

The tank entity itself could also be derived from a base entity class and look something like,

```
class Entity {       // Base class with entity identifier
  entity_id my_id;
};

class Tank : public Entity {
  int x, y;          // Its current position
  void loop(void) {  // Tank main code
    for (;;) {
```

[13]The operations do not constitute a full ECO API. First, a full API would also include an `unsubscribe` operation. Second, the `subscribe` operation would also include pre and post constraints. These have been left out, since they are not relevant for this thesis.

```
      // Receive input from player and calculate new position
      // Announce new position
      raise (new NewPositionEvent(my_id, x, y));
    }
  }
};
```

Minefield Entity

The minefield entity would subscribe to all events of the type `NewPositionEvent` which featured positions within the borders of that particular minefield. Assuming a minefield shaped as a square whose side is ten units long and whose bottom left corner is at $(90, 70)$, the code for the minefield could look something like,

```
class Minefield : public Entity {
  Minefield();
  void NewPositionHandler(entity_id, int, int);
};

Minefield::Minefield() {
  subscribe (NewPositionEvent,       // The event type name
             &NewPositionHandler,    // Handler for this subscription
             NC);                    // Notify constraint (see below)
}

void Minefield::NewPositionHandler(entity_id id, int x, int y) {
  // The notify constraint assures x and y are within the minefield
  if (random() < 0.1) { /* Boom! */ }
}
```

The notify constraint `NC` used in the above code can be expressed in many different ways, but it should insure that the event refers to a position within the minefield borders, i.e., evaluate,

$$(90 < x < 100) \wedge (70 < y < 80)$$

In the following sections we will extend the above example by considering different declarations of `NC`.

2.4.2 Approach 1: An Interpreted Language

The approach described in this section is based on notify constraints being strings containing expressions in some predefined expression language. Both of the CORBA Notification Service proposals described in section 2.2.2 use

such an approach. This section will extend the tank game from section 2.4.1 with an interpreted notify constraint. Since the tank game example is written in C++ we will assume the interpreted language uses C++ style expressions, but any language with equivalent expressive power could be used, such as either of the languages defined in the CORBA Notification Service proposals.

The following line of code extends the tank game from section 2.4.1 with a notify constraint in the form of a stringified C++ expression,

```
#define NC "90<x && x<100 && 70<y && y<80"
```

At runtime, any event of type `NewPositionEvent` would have to be matched against the above string. This is a problem, because events (as opposed to notify constraints) are compiled rather than interpreted. Hence, even though the `NewPositionEvent` class contains member variables named `x` and `y`, the information about the names of these members will have been stripped by the compiler during compilation.[14] The same is the case for type information. At runtime, there is no simple way to map the symbolic `x` in the above string to the corresponding event member variable and there is no way to match the type of the `x` in the string against that of the class member.

Specification and Evaluation Symmetry

The cause of the problem is that as part of an interpreted approach, we expect the interpreter to *evaluate* constraints that are really *specified* by the application. This is depicted in figure 2.4.

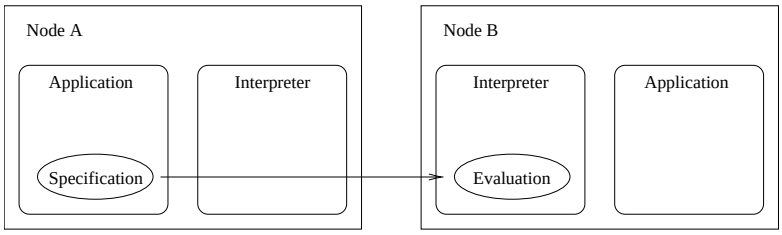

Figure 2.4: Asymmetric Specification and Evaluation of Notify Constraints

We assume the interpreter and the application to be distinct components and changes to the latter not to have any influence on the former. This is required such that new event types can be defined by the application. The consequence, however, is that the number and types of event parameters are

[14] Often compilers retain name information in the executable program to facilitate debugging, but this data is not readily available to the program itself at runtime and its format is highly compiler-specific.

not readily available to the interpreter unless explicitly specified. Hence, if we want the interpreter to match events against notify constraints, it needs to obtain this *event type information* somehow. Furthermore, notify constraints could likely (depending on their power of expression) require facilities for referring to event parameters. To match such a notify constraint against an actual event, the interpreter would need access to *name information* for event parameters. The problem is that events are really application-specific which means that evaluation of notify constraints (since notify constraints depend on event types) is really in the domain of the application as well. If we want to move evaluation away from the application and into the interpreter, we need to move type and name information along with it.

Having realized this, an obvious solution is not to move evaluation to the interpreter, but let it be part of the application instead. This is a more symmetric approach, and is illustrated in figure 2.5. This solution, however, requires the application to provide code for actually *evaluating* a notify constraint in addition to *specifying* the constraint itself. Also, this approach requires the interpreter to make an upcall to the application code as shown with the short arrow in the figure.

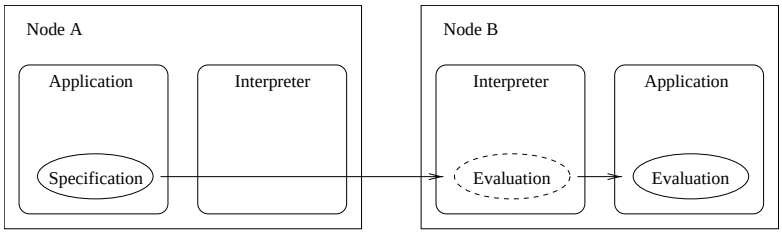

Figure 2.5: Symmetric Specification and Evaluation of Notify Constraints

Since name and type information for event member variables is specific to each event type, it makes sense to place code for matching constraints in event classes. In C++ this could be done by adding a virtual member function to the event base class defined in section 2.4.1,

```
class Event {
  ...
  virtual bool evaluateNotifyConstraint(string&) =0;
};
```

Events defined by the application would be derived from this class and would implement the above method such that it returned `true` or `false` depending on the result of the match. It thus becomes the *application's* responsibility to evaluate notify constraints whenever an event is raised. Effectively, this

means that the interpreter depends on the application to supply name and type information and evaluate the notify constraint instead of doing it directly. Moving the entire evaluation task to the application domain in this manner has one obvious drawback: it completely forfeits the goal of having an interpreter in the first place.

A more sensible combination could be to extend the approach such that the notify constraint is merely *passed through* the application in order for type and name information to be added and then evaluated by the interpreter. Such an approach is shown in figure 2.6. Arrow 2 shows an upcall from the interpreter to the application and arrow 3 shows the application invoking the interpreter with the type and name information added.

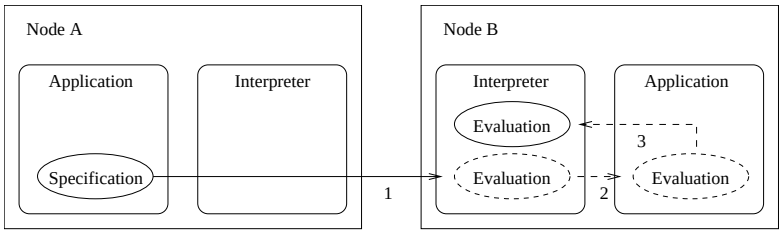

Figure 2.6: Passing Notify Constraints through the Application

The interface to the interpreter (whose invocation is shown with arrow 3 in the figure) need not be complicated. It could, for example, consist of a single evaluation function, taking type and name information in a string in a similar manner to the `vprintf` function known from C. It could for example look like,

```
#include <stdarg.h>

class Interpreter {
public:
  static bool evalNC(string& nc, string& format, entity_id id, ...);
};
```

The ellipsis dots (`...`) represent a variable number of arguments in common C++ fashion. Their types and names are given in the `format` string, for example as a sequence of tuples of $(name, type)$ in the following format,

$$\text{“}n_1 : t_1, n_2 : t_2, ..., n_i : t_i\text{”}$$

Each n denotes a name and t a type known to the interpreter. The values are given as `void` pointers that the interpreter can cast according to the type information in the format string.

Tank Game Extension

Recall our example from section 2.4.1 where we had tanks raising `NewPositionEvent` events as they moved around the world. Assuming that the previously defined notify constraint and event base class are still in effect, such an event, complete with wrapper method and constructor, could look like,

```
class NewPositionEvent : public Event {
  NewPositionEvent(entity_id, int, int);   // Event constructor
  bool evaluateNotifyConstraint(string&);  // NC evaluator function
  entity_id raiser_id;                     // The moved entity's ID
  int new_x, new_y;                        // Its new position
};

NewPositionEvent::NewPositionEvent(entity_id id, int x, int y)
  : raiser_id(id), new_x(x), new_y(y);

bool NewPositionEvent::evalNotifyConstraint(string& nc) {
  return(Interpreter::evalNC(nc, "id:entity_id,x:int,y:int",
                             &raiser_id, &x, &y));
}
```

Summary

This section has described how notify constraints can be expressed as strings subject to runtime interpretation. The name and type problems inherent in matching an interpreted constraint against a compiled event have been discussed, and we have shown how an application and an interpreter can work in tandem to solve these problems. The idea of passing the evaluation through the application to obtain type and name information may at first seem inelegant. However, since the wrapper method described above is indeed just that, it is an obvious candidate for automatic generation by a preprocessor. This means that applications could ignore the issue of evaluation altogether and concentrate on specification of the constraints alone.

The choice of language is, of course, paramount. To accommodate applications, a suitable language for specifying notify constraints would have to be designed. Such a language would need sufficient expressive power to accommodate application requirements. In particular, the set of types and operators would have to be chosen carefully, since it would not be extendable by the application. In this section we used C++ expressions. The languages described in the two CORBA Notification Service proposals are similar but solve the name and type problems by using positional notation (indexing members by their position) in conjunction with runtime type codes.

One unavoidable drawback of representing notifify constraints as strings subject to interpretation is decreased evaluation efficiency, because of interpretation overhead. On the other hand, this approach improves the possibilities for secure evaluation in the case of untrusted event subscribers. One could easily catch illegal operations (such as division by zero) and even impose a time or processor cycle limit on the evaluation of a particular constraint or all constraints from a particular source.

2.4.3 Approach 2: Event Templates

An approach described by [HBBM96] is to specify notify constraints (*acceptance expressions* in their terminology) in the form of *event templates.* When an entity subscribes to an event, it specifies a template for raised events to be matched against. As for the ECO model's notify constraints, the matching is done at the raising side and only if it is successful is the event propagated to the subscriber. The template approach of [HBBM96] is augmented with *side expressions* to increase the complexity of the constraints it is possible to impose. In this section we look at the 'pure' template solution as well as the one featuring side expressions. Both require the existence of an interpreter outside the application to match templates against events.

Description

Since the template language, unlike the language used to express notify constraints in section 2.4.2 (C++ expressions), cannot be assumed to be familiar to the reader, we present a description of it here before moving on to the tank game example. Abstractly, the event template approach means that if an event type has the form,

$$EventTypeName(arg_1, arg_2, ..., arg_n)$$

where $arg_1 ... arg_n$ are arguments given when an event is raised, then a notify constraint has the form,

$$EventTypeName(exp_1, exp_2, ..., exp_n)$$

where $exp_1 ... exp_n$ are expressions in some language, for example a modified form of regular expressions like that described in [Hay96].

Event templates of this form are called *base event templates* in [HBBM96] and can be used to form *composite expressions,* using the four composition operators *without, sequence, or,* and *whenever.* In addition, base event templates can be augmented with *side expressions* to impose extra constraints

on event arguments. A template with a side expression has the form,

$$EventTypeName(exp_1, exp_2, ..., exp_n)\{sideexps\}$$

where *sideexps* can contain expressions that have to be fulfilled in addition to the successful match of the template itself.

An example given in [HBBM96] is a printing service that raises events of a type $Finished(n)$ where n is a job number in form of an integer. An entity interested in the completion of print jobs can subscribe to this event using for example the template,

$$Finished(27)$$

if that particular entity is only interested in the completion of job number 27, or

$$Finished(*)$$

if interested in the completion of any print job. An example of a composite expression constructed from two base event templates could be,

$$Finished(27)|Finished(42)$$

which would cause propagation of either event when (or if) it occurred.[15] An example of an event template with a side expression given by [Hay96, p.67] is,

$$Withdraw(z)\{z > 500\}$$

where all withdrawal events (in a bank scenario) of more than £500 are propagated to the subscriber.

Tank Game Extension

Using the event template approach, our notify constraint from the section 2.4.1 tank game example could look like,

```
#define NC "NewPositionEvent(*,x,y){x>90,x<100,y>70,y<80}"
```

We need four side expressions to express the constraint. The wildcard character '`*`' matches the identifier of the moving entity, since the minefield is interested in anything that moves within its borders. The two variables `x` and `y` are bound by the interpreter during evaluation.

[15] Actually, the system described by [Hay96] returns the *set of occurrence times* where either event has occurred, and thus differs from the ECO model in this respect. For our purposes, however, assuming the event is propagated will do.

In addition, each event class would need code to transform the values of all relevant event member variables to a template. As described in section 2.3, this code can be generated by a preprocessor by reading an IDL[16] definition such as,

```
TankGame : INTERFACE =
  NewPosition: EVENTCLASS [ id : entity_id;
                            x : int;
                            y : int ];
END.
```

Discussion

The idea of notify constraints in the form of event templates is easy to grasp, but in its purest form—without composition and side expressions—it is very limited. Recall that events and templates have the form,

$$EventTypeName(arg_1, arg_2, ..., arg_m)$$
$$EventTypeName(exp_1, exp_2, ..., exp_n)$$

An event will match a template only if,

1. The event type names are identical.
2. The number of event parameters are the same, i.e., $n = m$ above.
3. The event parameter types match.
4. Each event parameter arg_i matches the corresponding template expression exp_i.

The last point is of particular importance. Since expressions are matched one by one against actual event parameters, there is no way for parameters to be compared against each other. For example, the constraint $arg_1 > arg_2$ cannot be expressed with pure templates, since a given exp_i is only matched against its corresponding arg_i. Furthermore, the pure template approach implies logical conjunction between the template expressions,

$$exp_1 \wedge exp_2 \wedge ... \wedge exp_n$$

Effectively, all expressions given in the template have to match for the evaluation to be true. It is not possible, for example, to specify constraints where it is sufficient for one of any number of expressions to match, i.e., of the form,

$$exp_1 \vee exp_2 \vee ... \vee exp_n$$

[16]Recall that Cambridge IDL is different from OMG IDL.

Though not stated explicitly, the author of [Hay96] has realized these limitations and augmented the template approach with side expressions to form a more powerful language for expressing constraints. In addition, templates can be used to create quite complex composite expressions.

Summary

We have described how notify constraints can be expressed as template strings subject to runtime interpretation in the manner proposed by [Hay96]. This approach solves the name and type problems described in section 2.4.2 in a very elegant manner by transforming event occurrences to strings and matching them against regular expressions style filters.

The actual expressive power is limited for pure templates, but can be augmented with side expressions as done by [Hay96] at the cost of some simplicity. The actual expressive power of the augmented templates is not discussed by [Hay96], but it is likely to correspond to an interpreted expression language featuring integers and strings as the only types.[17]

Evaluation efficiency was one of the design goals of the system designed by [Hay96]. Matching of text strings against regular expressions can be done in logarithmic time according to [BYG89] which means that the individual fields of *base templates* can be processed efficiently. The method for evaluating *composite events* is "designed to have an efficient implementation [Hay96][p.70]" but no estimate of speed is given. However, the algorithms could involve many operations that, depending on the rest of the design, could be expensive. No cost estimate of evaluating side expressions is given in [Hay96], but compared to native code this approach is bound to be more expensive in terms of processing resources.

2.4.4 Approach 3: Notify Constraint Objects

In section 2.4.2, we saw how the part of the evaluation that is application-specific, namely supplying type and name information, could be left to the application. In that approach, the rest of the task, namely evaluating the notify constraint was done by an interpreter for a language designed for that particular purpose. One could argue, however, that the interpreted language is superfluous since there is already a language available—the language in which the application is written. This section takes a closer look at an approach where code for evaluating constraints is generated before compilation of the application itself and compiled together with the rest of the application

[17] This assumption is based on the rather brief description of side expressions in [Hay96][p.67]; they are said to be "analogous to those found in formal methods."

code. It would be the kind of approach to use when implementing filters as event adaptors in the JavaBeans event model.

A possible way of avoiding the type and name problems described in sections 2.4.2 is to create code that evaluates the notify constraints and to link it into the application at compile time. The idea is to use a preprocessor to extract the notify constraint from the application code's `subscribe` statements and generate C++ classes complete with methods to evaluate the constraints. Code is also generated for creating a global table containing pointers to these objects. At runtime, this table is present on all nodes running the application, and it is thus possible to globally identify a notify constraint with an integer index into this table. The preprocessor also replaces the notify constraints given in the application's `subscribe` statements with plain indexes into the global table. At runtime, all notify constraints are present at all nodes in the form of a collection of objects linked into the application. When subscriptions are performed, a simple integer is passed rather than code for evaluating the constraint.

The Cambridge approach used the preprocessor to generate code that translated the event to a string (subject to matching against a filter by an interpreter). The approach described here uses the preprocessor to generate code for actually performing the match instead of relying on an interpreter.

Tank Game Extension

Let us have a look at our minefield constructor from section 2.4.1. Before being run through the preprocessor, the constructor would probably look something like,

```
#define NC "90<x && x<100 && 70<y && y<80"
Minefield::Minefield() {
  subscribe (NewPositionEvent,          // The event type name
             &NewPositionEventHandler,  // Handler for this subscription
             NC);                       // Modified notify constraint
}
```

The code produced by the preprocessor would look something like,

```
class nc {
  virtual boolean eval(...) =0;
};

class nc_0 : public nc {
  bool eval(int x, int y) {
    return ((x>90 && x<100 & y>70 && y<80) ? true : false);
  }
```

```
};

nc global_table[] = { &nc_0 };

void Minefield::Minefield() {
  subscribe (NewPositionEvent,          // The event type name
             &NewPositionEventHandler,  // Handler for this subscription
             0);                        // Index into global_table[]
}
```

Discussion

One obvious advantage that this approach has, compared to the two others, is speed. Since the code that evaluates notify constraints is compiled and not interpreted, it can be expected to run much faster than equivalent code in the other two solutions. Also, the ability to identify notify constraints with integers is good, since it means that little information has to be transmitted between nodes and this information (one single integer) is easy to marshal and unmarshal. In addition, type checking of notify constraints is done in the subsequent compilation of the generated code, completely eliminating the need for runtime type checking.

There is one problem with this approach though, namely that code for the constraints have to be generated at compile time. This means that unless the preprocessor can do a very advanced code analysis, all information about constraints has to be complete and available at event compile time. It is not possible, for example, for an entity to calculate its preferred constraints during its initialization phase and perform the subscriptions accordingly. A likely modification of our minefield example from section 2.4.1 would be to provide the constructor with the minefield coordinates, making the creation of new minefields more elegant. Assuming the existence of a type `string` with the concatenation operator `++`, the improved minefield constructor could look something like,

```
void Minefield::Minefield(int x1, int y1, int x2, int y2) {
  string NC = x1 ++ "<x<" ++ x2 ++ " && " ++ y1 ++ "<y<" ++ y2;
  subscribe (NewPositionEvent,     // Event type
             &NewPositionHandler,  // Handler to process events
             NC);                  // Notify constraint
}
```

This way of implementing the minefield is not possible with the compiled approach, since the notify constraint string cannot be evaluated by the preprocessor at compile time. Hence, the game could allow new minefields to

be added in mid-game (for example if tanks were allowed to lay out mines) but it would not be possible to implement with the preprocessor approach.

The existence of a global table makes it easy to pass notify constraints between nodes as part of subscription statements, but it has some drawbacks as well. Most important is the problem that the table is static and generated at compilation time by the preprocessor. This means that all notify constraints have to be known in advance and that new ones cannot be generated on the fly, unless explicit support for discovering and dynamically linking new constraints in provided. Also, it is not strictly necessary to store all notify constraints on all nodes, but only those that are effectively in use. The static approach is essentially *eager* propagation of notify constraint evaluation code, and for very large applications that use many different notify constraints this approach may prove unscalable. In this case, a dynamic (and *lazy*) propagation scheme is preferable.

Runtime Compilation and Runtime Type Information

A way of solving the problem with runtime-generated notify constraints could be to use a *runtime compiler* instead of a preprocessor. Runtime compilation lets a program, such as the minefield code shown above, compile and link code at runtime. PS-Algol [CADA87] is a language with support for runtime compilation in form of a library function that takes a source code (in form of a string) and returns a procedure. In the context of notify constraints, such a function could be used to map notify constraint strings generated at runtime to executable code.

A different approach could be borrow from object technology, such as CORBA. CORBA includes an Interface Repository (IR) which is essentially a register with type information. This information is available to the application at runtime and can be queried whenever the application encounters unknown types. Even in a system using compiled constraints, this means that new events and notify constraints could be introduced at runtime.

2.4.5 Summary

A central theme of this section has been the intepreted versus the compiled approach. In general, the concept of interpretation as opposed to compilation represents a dynamic instead of a static approach. This dynamism opens a number of very interesting possibilities, but they come at a price.

The Benefits of Dynamism

Composition If notify constraints can be generated on the fly, they can be generated automatically. This has been shown by [Hay96] to be sufficient to support *composition* of event templates; something that is currently not used in the other models. Though another notion of composition could perhaps be developed, the scheme described by [Hay96] has not only been designed but also implemented.

Scalability If notify constraints can be transferred between nodes and be cached as long as they are needed, it is not necessary for an application to hold all notify constraints at all nodes at a given point in time. If the application features a very large number of entities with a very varied selection of notify constraints, the dynamic approach may be much more scalable than the static. Our minefield entity from section 2.4.1 is a good example of this; no two minefields would share the same notify constraint.

Security We have not explored the issue of security in detail, but it seems obvious that interpreted languages offer better oppertunities for security control than the execution of native code. Interpreted languages can theoretically be made secure; Java is an example of such a language, even though current interpreters do not implement it securely.

The Cost of Dynamism

The strengths mentioned above of course do not come for free. The cost falls in two groups.

Expressive Power Interpretation relies on a statically defined set of types and operators that can be used in expressing notify constraints. Though this set can be made arbitrarily large, it cannot be extended by the application on the fly. In particular, none of the interpreted languages discussed in this chapter allow for the definition of new types and operators, so in this sense, the interpreted approach is weaker than the compiled. If the set of types and operators is chosen with care by the language designer, it may prove sufficient for applications, even though it is not as versatile as a real programming language.

Efficiency Interpreted evaluation of notify constraints is bound to be less efficient than evaluating one that is compiled into native code. This is perhaps the biggest drawback of the interpreted approach.

Hybrid Approaches

Runtime compilation and runtime type information makes it possible to benefit from some of the characteristics of 'pure' dynamism without being subject to all of the costs. Both enable an application to introduce new constraints at runtime and can therefore be used to support composition. Also, such constraints can be loaded on demand and they therefore scale better (with regards to size of the application executable) than a static approach. On the other hand, constraints linked at runtime are indeed still compiled and therefore cannot be subject to the same runtime security checks as an interpreted constraint. Expressive power would be equivalent to that of other compiled constraints and therefore stronger than those of typical interpreted constraint languages. Efficiency could be better or worse than a purely dynamic approach, depending on the speed of the runtime linker (and for constraints compiled at runtime, also on the speed of the compiler) compared to the number of times the constraint is used afterwards.

2.5 Summary

We have reviewed three event models which are different in many respects. The CORBA and Cambridge models share some similarities. Both are designed for large-scale systems and both have excellent filtering support. Since filters take the form of strings containing expressions subject to interpretation, dynamic linking of filters is not an issue. Dynamic linking of other types of code could be implemented, though, depending on whether it is possible in the implementation language. In comparison, the JavaBeans model is well suited for centralised or small-scale distributed applications but has no inherent support for filtering. As opposed to the two other models, the fact that it is tied to the Java language guarantees that dynamic linking of any type of code will be easy to implement.

This chapter also looked at different approaches to expressing filters. We identified and explained the important problem of managing type and name information for an interpreted language, and discussed the strenghts and weaknesses of each of three approaches.

In the next chapter, we will look at a fourth event model, ECO, which is comparable to the Cambridge model in terms of complexity. It is similar to both the CORBA and the Cambridge models in two ways. First, it has inherent filtering support but, as opposed to the other two, does not specify a filter expression language. Second, it makes no assumptions about the implementation language and whether dynamic linking is possible.

Chapter 3

The ECO Model

In this chapter, we take a look at the ECO model from a theoretical perspective. The discussion will place particular emphasis on the aspects relevant to notify constraints. In particular, we do not discuss the other types of constraints (*pre, post* and *synchronisation*) and we do not discuss *zones* as described by [O'C97]. The ECO model is described in detail in [SCT95] and [ODC+96].

First, section 3.1 provides an overview and a discussion of the ECO model as such. Then section 3.2 looks at a set of simple but pure (i.e., free from implementation considerations) algorithms in an attempt to understand and illustrate the data flow involved in each of the ECO operations. Section 3.3 looks at the two previous implementations of the ECO model, and section 3.4 sums up.

3.1 Overview of the ECO Model

The acronym ECO stands for *events, constraints* and *objects* which are the three central concepts in the event model used in the Moonlight project. In addition to these concepts, the model also defines an API with three operations. The following sections explain the concepts and the API and finally present a minimal scenario in order to relate the concepts and API to each other. The ECO model was originally designed for virtual world support and though the model is a general-purpose event model, the terminology still reflects the original domain. Hence, when we refer to an 'ECO world' we mean any domain where the model is used: virtual world, telecommunications, or otherwise.

3.1.1 Events, Constraints and Objects

The initial letters of the model's three central concepts form the abbreviation ECO. We will look at them here in the order *objects, events* and *constraints*.

Objects (aka Entities)

Everything in an ECO world is an *object*. Objects have methods and attributes, but cannot access each other's attributes or directly invoke each other's methods, i.e., they are encapsulated. Instead they communicate by sending and receiving events. Objects have identifiers that are unique within an ECO world.

ECO objects are often referred to as *entities* to avoid confusion with the objects known from object-oriented programming languages. In the rest of this document we will support this tradition, in particular because our implementation language is C++ where the term *object* is used differently. In practice, an ECO entity would often be implemented as an object in some programming language, for example C++, but using the same term would cause too much confusion.

Events

Events are the means of communication between entities. In the event-based programming paradigm, entities do not invoke each other's methods. Instead, they announce events that may, or may not, lead to invocation of other entities' methods. There are no other means of communication between entities. Events are typed.

Constraints

Constraints form a means by which objects can impose restrictions upon events they are interested in. As described in [SCT95] and [ODC+96], different types of constraints can be used for various purposes. Examples mentioned are to implement synchronization within the receiving object, to fulfill real-time requirements, and to control the propagation of events. This project deals only with the latter kind: constraints upon the propagation of events. This type of constraint is used by entities to specify which events should be propagated to them, i.e., which events the entities wish to be *notified* about. For this reason, they are called *notify constraints*.

3.1.2 ECO Operations

The event-based communications paradigm is different from RPC or message-based communication in that there is no specific receiver. Events are conceptually broadcast by the supplier, and it is up to each individual consumer to decide whether it is interested in a particular event. The ECO model specifies three operations (constituting the ECO API) that entities use to communicate.

Subscribe

The *subscribe* operation is used by an entity to register its interest in events. It has the form,

$$subscribe(eventType, entityID, handler, notifyConstraint)$$

The event type is the type of events the entity is interested in, the entity identifier globally identifies the subscribing entity, the handler is a callback method to be invoked when events are received, and the notify constraint is an optional filter which events are to be matched against. Any events raised after the subscription has been performed will be matched against the constraint (if any) and (subject to the result of the evaluation) delivered to the entity by invocation of the handler method. Such a method is called an *event handler* in ECO terminology.

Raise

The *raise* operation is used by an entity to generate an event. It has the form,

$$raise(eventType, eventParameters)$$

The event is delivered to all receivers that have registered interest in events of the particular type, subject to evaluation of their respective notify constraints.

Unsubscribe

The *unsubscribe* operation is used by an entity to cancel a subscription previously made with the *subscribe* operation. It has the form,

$$unsubscribe(eventType, entityID, handler)$$

3.1.3 Example of an ECO World

This section describes a scenario with a very small ECO world in order to relate the ECO concepts and operations from the previous sections to each other. Consider a virtual world with three entities A, B, and C as shown in figure 3.1. A subscribes to events generated by B and C, receives an event and then unsubscribes.

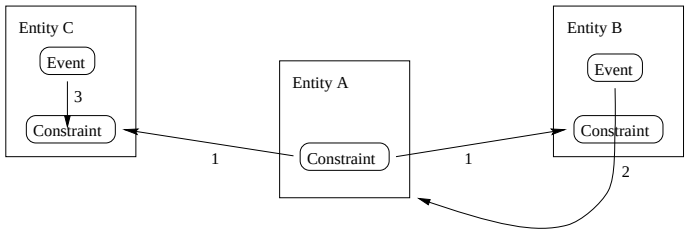

Figure 3.1: A Sample ECO World

There are four phases in the example: one subscription, two raised events and one unsubscription. The code using the ECO operations would look something like the pseudo-code shown below. The numbers refer to the arrows in figure 3.1.

```
1: entityA.subscribe(theEventType, entityA, myHandler, Constraint)
2: entityB.raise(theEventType, someParameters)
3: entityC.raise(theEventType, otherParameters)
4. entityA.unsubscribe(theEventType, entityA, myHandler)
```

Note that the steps could be executed either by multiple threads or a single thread. This is not important, as long as the steps are executed in sequence, so for simplicity they are shown as a single piece of code. There is only one event type involved, and two events of this type are raised. This happens in steps 2 and 3 respectively, where the two events are distinguished from each other by having different parameters. The code shown is pure ECO code and is therefore not concerned with the propagation and evaluation of the constraint. Behind the scenes, however, more complicated tasks are performed. This is explained below.

1. During the subscription phase, A passes a notify constraint object to entities B and C. This is depicted with the two arrows marked 1 in figure 3.1.

2. B raises an event which is matched against A's constraint. The evaluation returns true which means the event is delivered to A. This is depicted with arrow 2 in the figure.

3. C raises an event which is also matched against the constraint. This evaluation returns false which means the event is not delivered to A. This is depicted with arrow 3 in the figure.

4. A cancels the subscription. This removes the constraint from B and C. This is not shown in the figure.

3.2 Abstract Algorithms for ECO Operations

This section describes a set of algorithms for managing notify constraints and event subscriptions. Our approach will be to give simple and abstract versions of the algorithms to illustrate the flow of information necessary to manage subscription information. The algorithms are rather abstract and can appear naïve but they are not designed to be implemented in the form given, but rather to illustrate what tasks are necessary to manage event subscriptions and filtering. It is woth noting that application entities are not supposed to know anything about these algorithms; they invoke them exclusively via `subscribe`, `unsubscribe` and `raise` statements. The standard ECO terminology is used, meaning that,

- There are two parties involved in these algorithms and the term *side* is sometimes used to distinguish between them. The term *raising side* is used for the entity raising an event, and *receiving side* and *subscriber side* for the entity receiving it. It is not important whether the tasks described in this section are actually performed by the entities themselves or by 'somebody else' as long as it takes place on the same *side*, i.e., node in the network.

- An *event type* is defined by the name of the event class (assuming the implementation language has classes) and the types of the event parameters.

- The term *event* is used to refer to an event, including any parameters it may have.

- The term *handler* is really an identifier of a handler rather than the actual handler code itself.

3.2.1 Subscribing

The following algorithm is used when an entity subscribes to events of a particular type using the `subscribe` statement. It is executed on the subscriber

side. The algorithm data structures and flow are illustrated with an example scenario in figure 3.2. The close-up section of the figure takes place on the raising side and is explained in section 3.2.3.

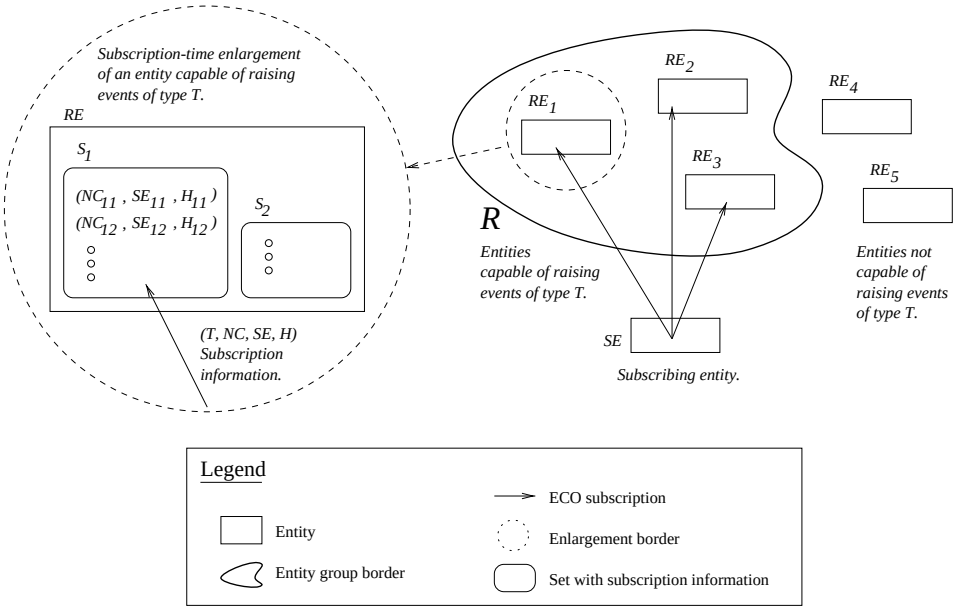

Figure 3.2: Subscription Data Flow

Algorithm 1 *Subscribing to events.*

1. *Let T be the type of event being subscribed to, NC the notify constraint, SE the subscribing entity and H the handler to be called when a matching event occurs. The tuple (T, NC, SE, H) constitutes subscription information.*

2. *Compute the set R of entities potentially capable of raising events of type T.*

3. *For each entity $e \in R$, propagate the tuple (T, NC, SE, H) to e as a subscription request.*

Note that the handler H is part of the subscription request because the *raise* operation described in section 3.2.3 requires access to it at a later stage.

3.2.2 Unsubscribing

This algorithm is used when an entity subscribes to events of a particular type using the `unsubscribe` statement. It is executed on the subscriber side.

Algorithm 2 *Unsubscribing to events.*

1. *Let T be the type of the event being unsubscribed, SE the subscribing entity and H the handler to be invoked. The tuple (T, SE, H) constitutes unsubscription information.*

2. *Compute the set R of entities potentially capable of raising events of type T.*

3. *For each entity $e \in R$, propagate the tuple (T, SE, H) to e as an unsubscription request.*

3.2.3 Raising an Event

This algorithm is used when an event is raised to determine to what entities the event should be propagated. It is executed on the raising side.

Algorithm 3 *Filtering events at the raising side.*

1. *Let RE denote the entity raising the event and T the type of event raised.*

2. *Let n be the number of different event types that can be raised by RE.*

3. *Let $S_1 ... S_n$ be a sequence of (possibly empty) sets containing tuples of the form $(NC_{i,j_i}, SE_{i,j_i}, H_{i,j_i})$ where $i \in [1; n]$ and $j_i \in [0; \infty[$. Each tuple represents subscription information for one entity subscribing to events of a particular type where $SE_{x,y}$ is a subscribing entity, $NC_{x,y}$ is the notify constraint associated with that subscription and $H_{x,y}$ is the subscribing entity's handler.*

4. *Let $\Phi(Y)$ be a one-to-one function mapping any given type Y to one of the sets $S_1 ... S_n$.*

5. *Let Q be a set of tuples of the form (SE, H) identifying the receivers of the raised event and their handlers. Initially, set $Q = \emptyset$.*

6. *For each tuple $(NC, SE, H) \in \Phi(T)$, evaluate NC in the context of the event raised. If the evaluation returns true, add (SE, H) to Q.*

7. *Propagate the raised event to all handlers h of entities e where $(e, h) \in Q$.*

3.2.4 Handling Subscribe Requests

The following algorithm is used when an entity subscribes to events of a particular type using the `subscribe` statement. It is executed on the side receiving the subscription.

Algorithm 4 *Handling requests for subscription.*

1. *Assume the existence of the mapping function* Φ *and the sets* $S_1 ... S_n$ *from algorithm 3.*

2. *Assume the incoming subscribe request has the form* (T, NC, SE, H) *where* T *is the type of the event,* NC *is the notify constraint,* SE *is the subscribing entity and* H *is its handler.*

3. *Add the tuple* (NC, SE, H) *to the set* $\Phi(T)$.

3.2.5 Handling Unsubscribe Requests

The following algorithm is used when an entity cancels its subscription to events of a particular type using the `unsubscribe` statement. It is executed on the side receiving the request for unsubscription.

Algorithm 5 *Handling requests for unsubscription.*

1. *Assume the existence of the mapping function* Φ *and the sets* $S_1 ... S_n$ *from algorithm 3.*

2. *Assume the incoming unsubscribe request has the form* (T, SE, H) *where* T *is the type of the event,* SE *is the unsubscribing entity and* H *its handler.*

3. *Remove all tuples* $(*, SE, H)$ *from* $\Phi(T)$ *where* $*$ *is any notify constraint.*

It is worth noting that the approach given above removes all matching subscriptions which uses the handler in question, regardless of which notify constraint was involved. As we shall see in section 4.1.1, there are other ways of doing this. This one is called the *wholesale* approach.

3.3 ECO Implementations

There have been two previous implementations of the ECO model. This section briefly describes them and points out the ways in which they differ from this one. Also, references to more detailed descriptions are given.

3.3.1 The VOID Shell (1995)

The VOID Shell [Tea95] is a system for non-distributed virtual world support. It includes several components, such as graphics support and an entity editor, including *ECOlib* which implements a version of ECO with notify, pre and post constraints. ECOlib is centralised (i.e., runs only on one host) and does not include facilities for dynamic linking of class code. The VOID Shell did not attempt to estimate the value of notify constraints.

3.3.2 DECO (1997)

[ODC+96] implements DECO (Distributed ECO) which relies on the ISIS framework for group communication. It features precompiled notify constraints which are not dynamically linkable and an extension to ECO called *zones*. The concept of notify constraints goes hand in hand with the concept of zones. In [ODC+96], zones are used to separate the virtual world into manageable groups of objects. Notify constraints are used within zones to limit the number of events communicated between objects in a particular zone. Effectively, zones provide an outer level of scope, whereas notify constraints provide scope on an inner level. [ODC+96] estimated the value of zones but not of notify constraints.

3.4 Summary

This chapter has described the ECO model thoroughly in an attempt to give the reader sufficient background to understand the rest of the thesis. An overview of the model was given before a set of simple algorithms to illustrate the way the model works. The most important concepts in the model (events, constraints, objects) were explained, and ECO terminology sufficient to understand the algorithms was given. Also, the chapter briefly described the two previous ECO implementations, in particular in the context of notify constraints.

Chapter 4

Analysis and Design

Of the three objectives given in section 1.6, *Objectives A* and *B* deal with the nature of the system we want to build. This chapter addresses these two objectives by analysing the problem domain. *Objective C* regards evaluation and will be addressed in chapter 7. To fulfill *Objectives A* and *B,* the system should have the following properties,

1. It should implement the ECO model as described in section 3.1.
2. It should be distributed.
3. It should implement notify constraints which are dynamically linkable.

Thus, the problem domain separates nicely into three areas, corresponding to each of the above properties. In the next three sections (4.1 to 4.3) we will look at each area in turn and list the problems in it. For each problem, we will discuss possible solutions and choose which one to use in the implementation. Finally, in section 4.4, we sum up the decisions and give an overview of the resulting system.

4.1 ECO Problems

Even though the ECO model as described in section 3.1 is fairly simple, there are some issues that were not addressed by the designers of the model but which an implementation still has to take into account. Three such ECO-related issues are dealt with in this section.

4.1.1 Method Bindings

When an entity in the ECO model subscribes to an event, it effectively binds the occurrence of *a set of possible events* to a particular method; all subse-

quent occurrences of these events are reported to the method in question. Multiple handlers can be bound to the occurrence of the same events; thus a single event may cause invocation of multiple methods in a single entity. This is depicted in figure 4.1. Bindings can be changed by *subscribing to* and *unsubscribing from* events. The former creates a binding and the latter removes it. The idea of notify constraints is closely linked to that of binding certain event occurrences to entity methods. Notify constraints provide a way of specifying the set of events that are reported to the entity.

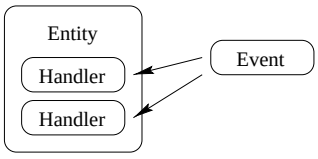

Figure 4.1: One event causing invocations of two handlers.

Multiple Subscriptions Using One Handler

As mentioned, figure 4.1, shows the occurrence of a single event bound to several handlers. With regards to the opposite, associating multiple subscriptions with the same handler (depicted in figure 4.2) the semantics are not as clear. The ECO model documents neither allow or disallow this use of the API.

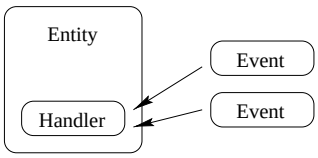

Figure 4.2: Two events causing two invocations of the same handler.

The problem stems from the fact that the tuple,

$$(eventType, entityID, handler)$$

specified as part of an unsubscription contains less information than the one specified at subscription time,

$$(eventType, entityID, handler, notifyConstraint)$$

The missing notify constraint means that an unsubscription cannot match a particular subscription. Another aspect of the same problem is whether to

allow multiple (identical) subscriptions to use the same handler and, if this is allowed, how to handle unsubscriptions. For an ECO implementation, there are four ways of dealing with this problem,

1. Enforce a one-to-one mapping implicity by requiring applications to implement exactly one handler for each notify constraint and prohibiting the use of the same handler for different subscription requests.

2. Allow the mapping of multiple (identical, or only with different notify constraints) subscriptions to the same handler, but accept the fact that once performed, an individual subscription cannot be cancelled. Rather, the unsubscribe request works as a wholesale operation, cancelling all that entity's subscriptions of that particular event type associated with the handler in question.

3. Extend the *unsubscribe* operation to match the subscribe operation by adding the missing notify constraint as a parameter. This also makes it possible to solve the problem for completely identical subscriptions in an elegant manner. Each unsubscription could simply cancel one subscription—exactly which one does not matter since they are identical.

4. Extend the *subscribe* and/or *unsubscribe* operations such that an individual subscription can be cancelled by letting the *subscribe* operation return a subscription identifier that can be passed as a parameter to *unsubscribe*.

All four approaches are plausible and perfectly possible to implement, and the ECO model does not specify whether one is more correct than the other. It is worth noting that even in the ECO documents there is some confusion as to how many arguments *unsubscribe* takes. For example, some sources (e.g., [Tea95, p.53]) do not use the *handler* parameter, whereas others (e.g., [O'C97]) do.

Decision

Of the four solutions listed, solution 1 seems unattractive because it imposes an unnecessary restriction upon the application's use of the ECO API. Solution 2 is better but the wholesale semantics complicate the model slightly and require more awareness from application programmers. Solution 3 is perhaps the best from an overall viewpoint, since it retains flexibility as well as simplicity, but it requires yet another change to the ECO API. Solution 4

also solves the problem, but in a more complicated way because it introduces a whole new concept, namely the subscription identifier, into the API.

In order not to change the standard ECO API, we decide upon solution 2, even though it complicates the application's view of the model slightly. We have, however, identified and documented the problem, and can make this peculiarity clear to application programmers.

4.1.2 Where to Evaluate Notify Constraints

One of the requirements of notify constraints in the ECO model is that it be possible to evaluate them without access to the entity that specified the constraint. This requirement makes it possible to propagate notify constraints to remote nodes and perform the filtering there. Effectively, the idea is to avoid propagating 'uninteresting' events to the subscriber and thus save communication resources.

However, the requirement of being able to evaluate notify constraints remotely means that only very limited data can be referenced, namely that which is accessible where the filtering is done. In the ECO model, only the actual parameters of any event raised are accessible, but one could also imagine access to the identifier of the entity raising the event or the identity of the physical node on which that entity resides. Of course, these are available at the raising side and could therefore simply be given as event parameters by the raising entity itself.

Delayed Filtering and Multicast

Even though the ECO model disallows access to the receiving entity's attributes to facilitate the remote evaluation of notify constraints, there is still the possibility of evaluating those constraints at the receiving side—or possibly both sides—meaning that events are allowed to be transmitted over the network *before* the filtering is done. Consequently, events may be discarded at the destination, after bandwidth has been spent transmitting them. At a first glance, this approach may seem to forfeit the goal of the ECO model's notify constraint concept as such. This is particularly true, if the model is not changed to allow access to the receiving entity's attributes even though they are now local to the evaluation. At a closer look, however, performing filtering at the receiving side may still make sense. The primary reason is the existence of *multicast* or *group communication* facilities present in some network protocols (e.g., IP and ATM, the latter in the form of multipoint calls).

A central characteristic of group communication is that it is fairly expensive to change group memberships, whereas multicasting a message to all members of an existing group is relatively cheap in comparison. If an entity at a given node raises a series of events that go almost (but not quite) to the same recipients, it may be more efficient to transmit the event to all receivers, including some that are not interested, rather than to repeatedly change group memberships.

Bundling

In the naïve algorithms presented in section 3.2, all entities communicate directly. An obvious improvement to these algorithms would be to *bundle* events, such that even though a particular event is destined for *two* entities on the same node, it would only be transmitted over the network *once*. This poses a problem, however, since it requires the receiving node to determine who are the local receivers of a particular event. Determining the set of local receivers for such an event involves either evaluating notify constraints for all potential receivers *at the receiving side* or including the identities of the receiving entities in the message. The latter obviously scales poorly, since the event may be multicast to a large number of nodes, each holding a large number of receiving entities.

Strategies

There seems to be three strategies for evaluating notify constraints,

1. At the source only. This was the original intention in the ECO model. As mentioned, the notify constraints do not have access to data at the node where the subscriber resides, precisely because it should be possible to evaluate them without access to the receiving entity.

2. At the destination only. This forfeits the overall goal of saving any bandwidth, since events will be broadcast to everybody. There is little to gain from this approach, except for a simplified programming model.

3. At the source and the destination. This imposes more processing overhead than the two other solutions, but would make sense if a multicast operation with a large group creation overhead is available. In this case, a series of multicast groups could be maintained and the event sent to the smallest group containing all receivers. This also makes it possible to bundle events and thereby reduce network traffic.

Decision

Solution 1 is the safe and traditional choice which falls within the 'ECO spirit,' but it will make it infeasible to use multicast in conjunction with event bundling. Solution 2 is uninteresting since it would make the system less scalable which is the exact opposite of what we are trying to achieve. Solution 3 may seem to forsake the original ECO design principles since it requires double evaluation of constraints, and therefore more processing overhead, but the evaluation takes place at different nodes and the load is therefore distributed.

The ability to benefit from multicast support is a very attractive characteristic for a distributed event system, and one we would not give up lightly. Since we will most certainly want to bundle events, the most sensible approach seems to settle for solution 3, even though it falls outside the original ECO design ideas.

4.1.3 Latecoming Entities

When new entities join a world that already exists, there may be subscriptions in place that the new entity does not know about. Such a scenario is shown in figure 4.3 where entity C enters a world where A and B have already exchanged subscriptions. C may be able to raise events that would match A's and B's old subscriptions, and should therefore be delivered to the two entities. However, having joined the world later than A and B, C has not received the original subscriptions and is not aware of its two peers' interests. This is a problem, because old subscriptions should also apply to new entities, according to ECO semantics.

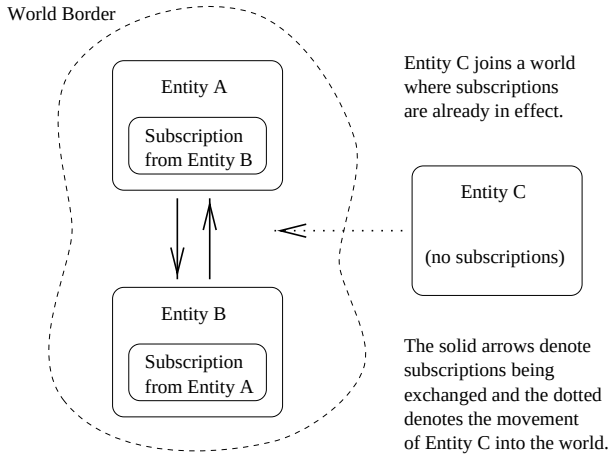

Figure 4.3: Entity C is a Latecomer.

The solution is that, somehow, C must obtain these subscriptions when it enters the world. The ECO model does not specify how, but there are two obvious approaches,

1. Keep all active subscriptions in a register (centralised or distributed) where new entities can obtain them.

2. Let each entity retransmit any active subscriptions it has when it learns about a new latecomer.

In conjunction with multicast groups, solution 2 could possibly be integrated into the group management protocol. For example, a new entity could multicast a *join group* style message to all other entities, causing each of them to send their active subscriptions to the new entity. This, however, still means that a lot of bandwidth would be spent retransmitting subscriptions. In addition, the new entity may easily be flooded if many subscriptions are retransmitted simultaneously.

Decision

Solution 1 described above is unattractive because centralised information services do not scale well and are subject to failures. A solution could be a distributed service, but building such a service would be well beyond the scope of this project. Also, there are obvious race conditions involved in keeping a register up to date. Solution 2 is attractive because it is simple to implement and does not suffer from race condition problems. However, it does not scale very well because it may potentially involve communication between many nodes. The best, and only really scalable, solution seems to be a distributed information service maintaining subscriptions. However, to limit the work involved, we settle for solution 2. Future versions of the system should take this into account.

4.2 Distribution

In this section, we present and discuss the problems related to distribution. The approach we are choosing is highly distributed, meaning that we have as few centralised components as possible. Essentially, there are only two, the Application Instance Register and a shared file system.

4.2.1 Architecture

Our architecture is based around the concept of *Application Instances*, or *AIs*. An application instance is a program running on a node in the network. It hosts a number of entities and relays events raised by them to other AIs. An AI also receives events from other AIs and relays them to its own entities. A scenario with six entities hosted by three AIs running on two nodes is shown in figure 4.4.

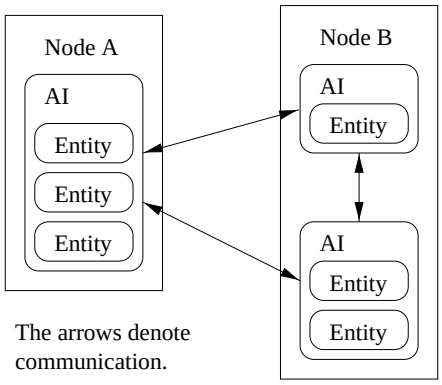

Figure 4.4: Scenarios with Nodes, Application Instances and Entities.

Implementation Strategy

The scenario shown in figure 4.4 is essentially a process group scenario. Group communication is a well-known concept from distributed systems theory, and therefore, an obvious solution is to use an existing package that implements group services. One such system, ISIS, was used by [O'C97] but was found to impose a fair amount of overhead, and not all of the ISIS features (such as the ordering facilities) were necessary.

Another approach could be to implement a minimal set of group communication facilities using a network level communications package. Such a system is described in the rest of this section.

Application Instance Register

When an application instance is created, it needs to obtain information about other AIs in the world. To do this, it uses a centralised register called the *Application Instance Register*, or *AIR*. This is a simple server program that maintains the authoritative list of AIs in the world. During its initialisation phase, a new application instance first reads the address of the AIR from a

file in a shared file system, then makes its presence known to the AIR (using a *record* type operation) and obtains a list of all the other AIs in the world (using a *list* type operation). When shutting down, the application instance notifies the AIR, using a *remove* type operation. This scheme requires a total of three message types to be exchanged between an AI and the AIR.

A file in a shared file system is used to hold the address of the AIR. This file is written by the AIR at initialisation time and subsequently read by each AI started.

Inter-AI Communication

During an AI's lifetime, it will need to communicate with peer AIs. There are three cases,

1. When a new AI enters a world, other AIs will need to be notified of its presence, so they can transmit future subscriptions and events to it.

2. When an ECO operation (*subscribe, unsubscribe* or *raise*) is performed by one of its entities, an AI may need to relay this operation to other AIs.

3. When an AI wants to leave the world, it should notify other AIs so they no longer attempt to relay data to it.

AIs need the ability to play both roles (sender and receiver) in each of the above cases. Note that cases 1 and 3 involve communication will all other AIs, i.e., the group of all peers. Depending on the availability of multicast on the underlying platform, this could be more or less expensive, and a scalable solution should use a multicast approach. Case 2 involves communication with a subset of this group and would also benefit from the availability of multicast. Since there are a total of three ECO operations (*subscribe, raise* and *unsubscribe*), the inter-AI communication will require a total of five types of messages.

4.2.2 Consistency and Ordering

Section 4.2.1 described two groups of messages, required for AI↔AIR and AI↔AI communication respectively. These messages are used to keep the different components of the system (the AIs and the AIR) up-to-date as to the state of the world. The order in which these messages are exchanged influences the different components' world views. In this section, we describe to what degree these views can be allowed to become inconsistent. The problem falls into two groups, AI management and event ordering.

AI Management

Managing the set of application instances currently in the world is the same problem as managing process group memberships. Since we are using the AIR to hold the authoritative copy of the list of application instances, we are using a centralised solution. Recall that the initialisation phase for an application instance is,

1. Register with the AIR.
2. Obtain complete list of AIs from AIR.
3. Register with all other AIs.

After this series of steps has been completed, the initialisation phase is over. Depending on the implementation of step 3, there may be temporary inconsistencies between the world views of the 'other AIs,' as they are notified. Consider the scenario where there are two AIs A and B in the world. Two new AIs C and D start executing the above steps simultaneously. If step 3 is implemented with unordered multicast or even as a sequence of unicasts, A and B may receive the registration messages from C and D in a different order, causing them to temporarily disagree on the set of AIs in the world. If events are raised on C and D during this interval, they will be delivered to different sets of entities.

If step 3 can be implemented with sync-ordered multicast [CDK94, p.339], no temporary inconsistencies will occur.

Event Ordering

When an event is raised by an entity, it is relayed by the AI hosting the entity to other AIs who again will relay it to the entities they are hosting. The event can be transmitted over the network from the source AI to the destination AIs in several ways, exactly like the registration request mentioned above. If ordered multicast is used, all receivers will agree on the sequence of events, but if unordered multicast or a sequence of unicasts is used, they may not.

Discussion

Whether strict ordering of events is important, depends on the application using the ECO implementation. As pointed out by [O'C97, p.87], the requirement to support strict consistency may become very expensive as the system scales. Also, implementing strict consistency without the use of an appropriate package (such as ISIS) would be a considerable effort. In this project, we

will therefore assume that the application does not require strict ordering of events and can accept that events, may be delivered inconsistently, i.e., in different order to different AIs.

If required, events could be serialised by the application itself, by implementing a *sequencer entity*. Such an entity would subscribe to all events, add a serial number and and retransmit them. Subscribing entities would then subscribe only to events raised by the sequencer entity. This is effectively a centralised implementation of totally ordered multicast. Because it is centralised, it will be a bottleneck as well as a single point of failure and it will not scale well. Algorithms for distributed sequencers exist; one is given in [CDK94, pp.608–612].

4.2.3 Concurrency and Synchronisation

One of the advantages of distribution is concurrency. The software components of a distributed systems run in parallel, independently of each other, on several levels. In the context of SECO, we have two cases; there can be several AIs running concurrently and there can be several active threads running within an AI. We call these two cases *inter-* and *intra-AI* concurrency and will look closer at them here.

Inter-AI Issues

Each application instance runs independently of the others. Though multiple application instances may be run on the same node, they will often reside on different nodes and communicate over the network. There are no means for application threads to be synchronised with threads hosted by other AIs.

Intra-AI Issues

An application instance hosts a number of entities which are created by the application while the AI is running. Some entities may be active and raise events with certain intervals as well as receiving events. Such entities are likely to require one or more threads to be running inside them. Other entities may be passive, performing subscriptions when they are created and thereafter waiting passively for events to occur. Such entities may not require threads.[1]

[1]The distinguishing between active and passive entities here is somewhat academic. There is no inherent difference between them, except for the presence of threads. An active entity can easily choose to become passive or vice versa.

Thread support is provided through the ROO library [ZC95] [Tay95] which implements threads as well as intra-process synchronisation primitives, such as semaphores.

4.2.4 Fault Tolerance

Distributed systems in general fail in much more interesting ways than centralised systems. When one or more components of a distributed system fails, the remaining components need to become aware that parts have failed and take some appropriate action. In this section we look at the four different failure scenarios which apply to this system. In general, our disposition is not go out of the way to make the system exceedingly fault tolerant, since this is an experimental project and high fault tolerance is not one of the objectives, as described in section 1.6.

Application Instance Failure

During execution, an application instance can fail because of bugs in the application, the SECO implementation or one of the toolkits. This situation is different from the node failure scenario below, because a node may host several application instances.

When an application instance fails, all entities that it hosts disappear. SECO entities are not persistent (unless the application implements some sort of persistence) and therefore disappear permanently if their hosting AI fails. Peer AIs discover that an AI has failed if sending messages to it fails and can then remove it from their records. The AIR does not initiate communication with AIs and will therefore not discover whether AIs have failed. This could be remedied by letting other AIs notify the AIR, but in order to limit the workload involved, we have chosen not to implement this functionality in the current version of SECO. This means that if a SECO environment is running over a period where application instances fail, the AIR will propagate obsolete entries to new AIs. However, these new AIs will remove the dead entries themselves, as they discover that they are no longer valid.

Node Failure

A node can fail for various reasons, for example hardware or operating system failures. In this case, all application intances running on that particular node fail as described above.

Network Fragmentation

Network fragmentation is a particularly complicated failure scenario in distributed systems, since it means the failure not of an actual system component but of the communications pathway between them. This means that different parts of the system may remain intact but believe that other parts have failed. Recovery of this situation is possible but fairly complex, and to limit the project workload we will therefore assume that network fragmentation does not occur.

AIR and Shared File System Failure

The AIR and the shared file system are the only two centralised components in the architecture and are therefore obvious single points of failures. An active SECO environment is very dependent on the availability of these two components and it would be attractive if they were highly resistant to failures. This could be achieved by using replication, both in the case of the AIR and the file system. Replicated file systems such as AFS [CDK94, pp.232–242] are hosted by several servers and is more resistant against failures than centralised file servers.

On the other hand, the AIR is a fairly simple program which relies on few other components and it should be possible to provide a fairly stable implementation. Also, NFS, the shared file system that comes with all versions of UNIX, is well-known technology and has been reasonably stable for a number of years. However, no matter how well-known and stable the software is, a centralised solution is always subject to hardware failures. In order to limit the project workload, we will choose the implement these two components as simple centralised components. If it is later shown to be necessary, replacing them with replicated services will be possible but will require extra work.

4.2.5 Scalability

In section 1.5 we identified four directions of growth of an ECO virtual world system: *users, entities, nodes* and *activity.* Since the SECO implementation is highly distributed, except for the AIR and the shared file system, any number of new *nodes* can be added. Also, new *users* and *entities* can be added, as long as there are nodes enough to support them. The remaining issue *activity* is the most critical and indeed the one the ECO model is designed to improve by using notify constraints. In chapter 7 we will try to evaluate to what degree this is true.

4.3 Dynamic Linking

In this section, we present and discuss the problems related to dynamic linking of class code.

4.3.1 What is a Dynamic Class?

A dynamically linkable class (called a *dynamic class* for short) is a class whose code can be linked dynamically at runtime. This is done by splitting the class code in two parts, *stub code* and *real code*. Each part implements the interface of the dynamic class (the attributes and methods) identically but instead of the real class implementation, the methods in the stub code contain code which links the real code by loading it from stable storage. Hence, the stub code is responsible for linking and invoking the real code in case it is ever needed. Since little code is required to perform the linking and relaying the invocation, the stub code will most often be substantially smaller than the real code. In this case, the footprint of the application containing the dynamic class will be reduced as long as the real code is not linked. The stub code has another feature called *versioning* which allows multiple implementations of the real code for a particular class to coexist. When creating an object, an application can ask for a particular version of the class code to be used. In this way, code can be incrementally added to an application as long as it implements a class interface for which stub code already exists in the application.

The Class Register

Support for dynamic classes is obtained by using a toolkit called the CLASS REGISTER [Fur97] which implements the functionality described above. Using this toolkit is complicated and involves a fair amount of overhead. First, each dynamic class must be registered with the CLASS REGISTER before it can be used, which involves the creation of various identifier objects. Also, in addition to the stub code and the real code described above, the CLASS REGISTER requires the application to supply upcall code for each dynamic class. This code is used by the CLASS REGISTER to access objects belonging to the dynamic class.[2] This means that each dynamic class requires four files with source code (header file, stub code, real code, and upcall code) whereas an ordinary (non-dynamic) class only requires two (header file and real code).

Along with the CLASS REGISTER comes a preprocessor, called DCLASS, which is supposed to transform non-dynamic classes into dynamic by gener-

[2] See [Fur97, p.38] for further details.

ating stub code and upcall code from a header file and a file with real class code. Unfortunately, the author of this thesis has found the preprocessor rather unstable and close to unusable for any but the simplest programs. There is no doubt, however, that it is possible to automate the process.

4.3.2 Making Class Code Available

The CLASS REGISTER requires the application to register each dynamic class before any instantiations of it are created. In [Fur97] this is called *class installation*. The parameters given to a class installation are all allocated and managed by the application. They are,

Numeric Language Identifier identifying the implementation language, such as C++.

Numeric Class Identifier identifying this particular class within the application.

Class Version Number is optional, but if versioning is used it identifies a particular version of the class. Multiple versions of the code for a dynamic class can coexist in the application. If versioning is used, a version number is specified when a class is instantiated.[3]

Symbolic Class Name is a text string containing the class name. There is a one-to-one relation between this name and the numeric language identifier mentioned above.

Size of Stub Code in bytes, as returned by the C++ `sizeof()` function.

Real Code Location is a text string containing the location (within a locally accessible file system) of a shared object file with the real class code.

Upcall Code Location is analogous to the real code location mentioned above but specifies the location of the upcall code.

In SECO, instantiations of dynamic classes (in the form of notify constraint objects) can be passed between application instances as part of subscription requests. Propagating such objects between application instances requires special attention because the application instance receiving the object must be able to install the dynamic class code locally.

[3] In practice, this is done by passing a versioning identifier to the C++ `new` operation by using the *placement* syntax described in [Str91, pp.497–499].

Remote Installation of Dynamic Classes

In order for an application instance to install a dynamic class, the following requirements must be met,

1. The receiving application instance must be linked with stub code for the class. This means that no new notify constraint classes can be introduced at runtime unless they share stub code with an existing dynamic class.

2. The seven parameters mentioned above must be available in order to use the CLASS REGISTER's install function.

3. The files containing real code and upcall code must be available so the code can be linked in case it is invoked.

The two last requirements are to the availability of certain information to the receiving application instance. An obvious way of making this data available would be to transmit it over the network along with the object itself. For the seven parameters mentioned in requirement 2, four integers and three text strings, this would not involve unreasonable overhead but the two shared object files from requirement 3 are different. They are potentially large files, and since they may never be used, sending them over the network could be a waste of bandwidth. A more attractive solution is to store them in a shared file system from which the receiving application instance can get them if it ever needs to.

Of the seven parameters listed above, the first four are sufficient to identify a class uniquely. This means that these four can be used to deduce a unique place in the shared file system where the remaining three can be stored. Whereas this does not decrease bandwidth usage (since the three parameters will be read from the shared file system anyway) it does simplify the format of the marshalled object transmitted over the network.

4.3.3 The Class Repository

As mentioned in section 4.3.1, using the CLASS REGISTER is fairly complicated. Apart from the amount of extra code (in the form of upcall and stub code) required, the installation of a dynamic class itself involves the creation of four identifier objects followed by invocation of the CLASS REGISTER itself. This, together with the fact that the CLASS REGISTER itself does not include a shared file system as described in section 4.3.2, provides incentive for building a front-end to the CLASS REGISTER. This front-end is called the *Class Repository*, or CREP.

Simplified Interface

One of objective of CREP is to let dynamic classes be registered with a minimum of effort from the application programmer. Therefore, the CREP interface is simpler than that of the CLASS REGISTER, consisting of a single function which can be invoked without the creation of identifier objects.

Storing Class Information

CREP relies on a shared file system to store class code and extra parameters in the manner described in section 4.3.2. The first four parameters are used for naming a directory (unique for each dynamic class) where the remaining parameters and the two shared object files are stored. Since we are already using a shared file system to store the AIR address (see section 4.2.1), it makes sense to extend that file system to also include the dynamic classes. Since multiple application instances can install classes simultaneously, CREP uses file locking to avoid race conditions.

4.4 Design Summary

Figure 4.5 on page 69 shows an overview of the system we have defined in this chapter. Two applications are shown in the middle, communicating through ECO libraries (SECO) and class repositories (CREP). The application instance register (AIR), described in section 4.2.1, and the different instances of SECO use a communications package to send and receive messages. The different instances of CREP use a shared file system to exchange class code used by the application. In addition, the AIR writes its address to the shared file system from where it is subsequently read by the SECO libraries. This is not shown in the figure in order to not complicate it further.

Below, we sum up the decisions in the previous sections. The implementation of the system is described in chapter 5.

ECO

We have identified and discussed three problems with regards to ECO implementation in general, and we have defined a version of ECO that solves these problems within the 'ECO spirit' and without substantial tradeoffs. It should therefore be easy to use this ECO implementation in conjunction with other ECO-related work, such as the extended ECO described in [O'C97].

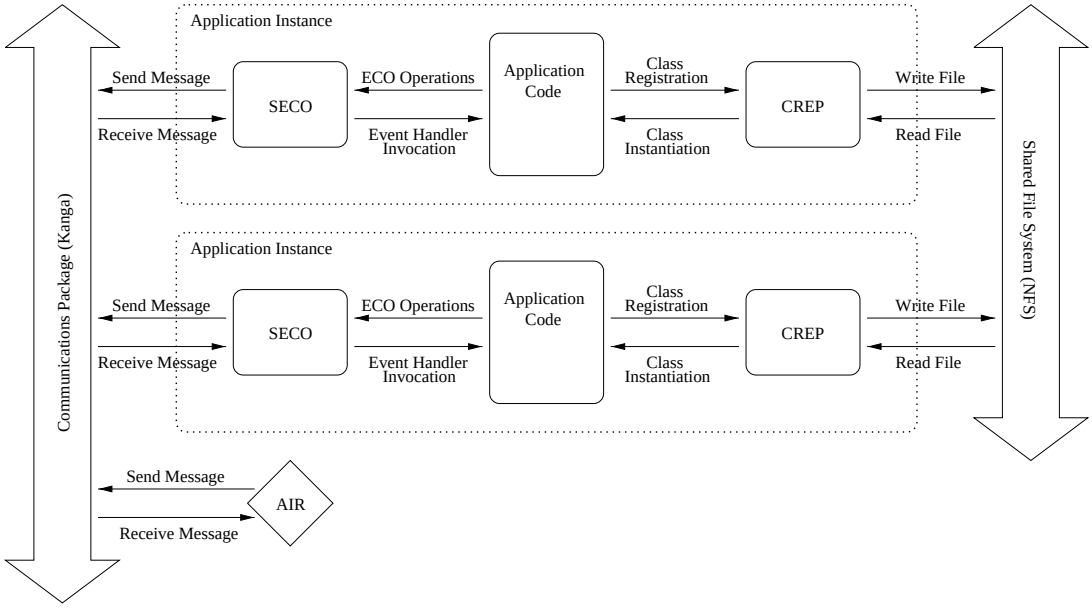

Figure 4.5: System Overview with two AIs and the AIR.

Distribution

We have designed a highly distributed architecture with only two centralised components, and we have shown that these two could be replaced with equivalent distributed components. We have discussed the problems inherent in distributed systems and seen that strict consistency could be maintained through message ordering, but in order to keep implementation workload down, we chose to implement a lax approach where temporary inconsistencies are allowed. Also, we looked at concurrency and synchronisation issues in the architecture and described what features are available to application threads. With regards to fault tolerance, we briefly discussed the three possible failure scenarios, even though the ability to deal with failures is not a primary goal. At last, we looked briefly at scalability in the system and concluded that with the architecture presented in this chapter, scaling *activity* is the biggest remaining issue and one where notify constraints will be important.

Dynamic Linking

We have described a way of using the CLASS REGISTER to build a class repository CREP that, in conjunction with changes to the application code, can be used to link classes dynamically. The class repository CREP uses the

shared file system introduced in section 4.2.1 to store dynamic class code and other class information. Though installation of dynamic classes is easier with CREP than the CLASS REGISTER the class code changes required are fairly complex and imposes a substantial overhead on the application programmer. However, with the current CLASS REGISTER there is no way to decrease the amount of changes required. Since the process of transforming conventional class code to dynamic class code is subject to automation, one approach to remedy the problem could be to repair the preprocessor [Fur97] or to extend the existing ECO Entity Editor [Tea95] to produce code that is dynamically linkable.

Chapter 5

Implementation

This chapter describes the SECO implementation. Owing to the analysis in chapter 4, the programming task was fairly straightforward and no serious problems were encountered. Consequently, this chapter does not raise any major issues not covered in the preceding chapter.

The system consists of nine components which can be grouped into three conceptually distinct domains or layers. Figure 5.1 on page 72 shows an overview of the implementation with the three layers marked and code usage represented by arrows. Arrows pointing downwards represent normal invocations whereas those pointing upwards signify callback style invocations. The upper layer contains only one component: the application using SECO. The middle layer, SECO, contains five components: event manager, communications manager, constraint manager, class repository and application instance register. The lower layer, the support layer, contains three components: ROO, KANGA and the CLASS REGISTER.

All components in middle layer are the original work of the author, and in addition, modifications to one of the components in the support layer (namely the CLASS REGISTER) were made in order to adapt it to SECO use. In the following sections, we look at each of the components in figure 5.1 from the bottom-up. This order makes sense because the components towards the bottom of the figure provide a foundation for those above them. Section 5.1 describes the support layer components, section 5.2 the SECO components. The top layer is not described in this chapter but in section 6.2 in form of a sample application. Section 5.3 concludes the chapter.

As can be seen from figure 5.1, the application layer uses three other components (ROO, the event manager, and the class repository), and an application programmer should therefore be familiar with the APIs of these three components. They are described in section 5.1.2, 5.2.4, and 5.2.1, respectively.

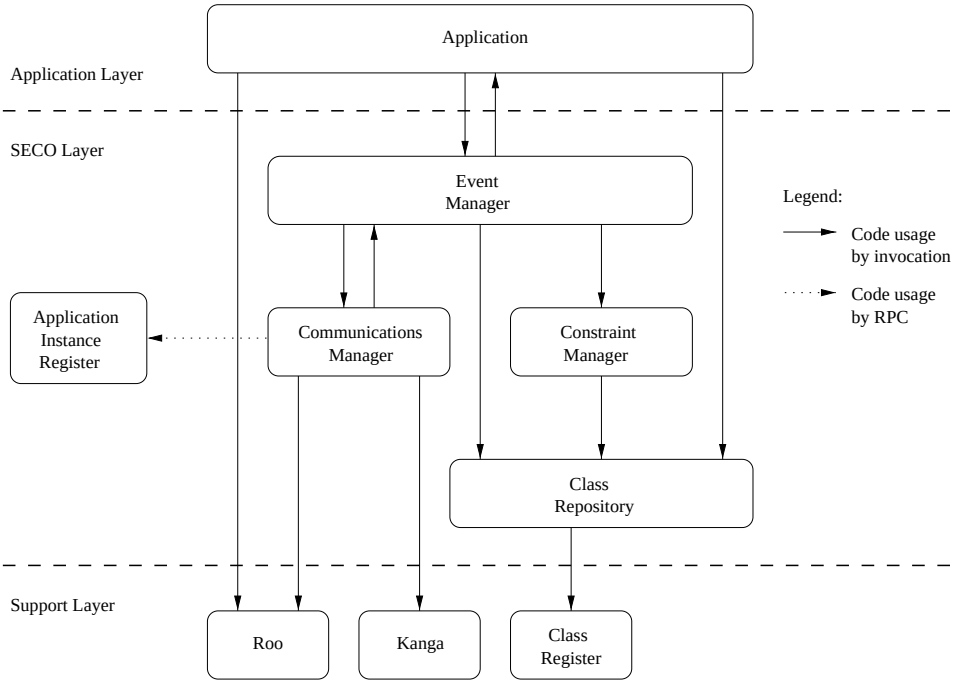

Figure 5.1: Implementation Overview

5.1 Support Components

This section describes various support components which the implementation relies on. Most of them were developed at Trinity College as part of the TIGGER project.

5.1.1 Kanga for Communications

KANGA is part of the TIGGER framework and provides a convenient class-oriented front-end to the transport layer (TCP/IP) based on *connection end-points* rather than hostnames and ports. Unlike TCP, KANGA is message-oriented: Data is *pushed* into messages, sent across the network and then *popped* at the destination. Marshalling operations for all standard C++ types are included. Unfortunately, the current version of KANGA does not have multicast capabilities, even though it is supported by the underlying operating system. KANGA relies on ROO (see section 5.1.2) for multi-thread capabilities.

The implementation uses the KANGA [Bur96] library for network communications support and the KANGA name service for registration of the Application Instance Register. Therefore the application must be linked with the

`libkanga.a`, `libtigger-status.a` and `libtigger-trace.a` libraries.

5.1.2 Roo for Multiple Threads

The ROO [Tay95] library implements multi-thread support within a single Unix process. The library also includes thread synchronisation primitives, such as semaphores. It uses the two other TIGGER libraries *status* and *tracing* and these libraries must therefore be linked with the application.

The implementation uses ROO to support multiple threads in order to receive and process requests received over the network. The application code itself can use the ROO API (described in [Tay95]) if it requires multi-thread support. In both cases the application must be linked with the `libroo.a` library.

5.1.3 Class Register for Dynamic Linking

The CLASS REGISTER [Fur97] provides support for dynamically linkable classes. It consists of two components: the CLASS REGISTER itself and the DCLASS preprocessor. The preprocessor takes as input a normal set of C++ source files (a `.h` header and a `.cc` source file) with ordinary class code and outputs class code modified to be dynamically linkable. The original class implementation is not linked into the application but compiled into a `.so` shared object file. Replacement class code, generated by the preprocessor, is linked into the application instead. When (or if) this code is invoked at runtime, the replacement code dynamically links the real class code into the application and relays the invocation. The CLASS REGISTER supports class versioning, meaning that code can be linked incrementally. Unfortunately, the DCLASS preprocessor is not entirely reliable and the author found it impossible to make it work on the application code. Therefore, code was handwritten to conform to the standards required by the CLASS REGISTER instead. The DCLASS preprocessor was not used.

The implementation uses the CLASS REGISTER to implement the Class Repository described in section 5.2.1. The latter extends the former to include a shared file system where shared object files are stored.

Modifications

The CLASS REGISTER included in [Fur97] was in form of a series of object files which had to be linked with a sample application. The sample application was to some extent hardwired into the CLASS REGISTER and some effort was

spent by the author of this thesis separating the two and turning the CLASS REGISTER into a library.

5.1.4 Other Support Classes

Two other support classes were created specifically for this project but are of a general nature. One is a simple implementation of a boolean data type, since the compiler we are using (GCC 2.6.3) does not include support for the C++ `bool` type.

Smart Arrays

The other support class, which is used extensively in the SECO implementation, is an array template. It implements automatically expanding arrays not unlike those found in the Emerald [BHJ+87] programming language. If an array cell is referenced which is out of bounds, the array is expanded to include the indexed cell. In this case, new objects are constructed and put in the array. An array can be queried for its current bounds, so iteration can be performed on the elements.

The current implementation of the smart array is not particularly efficient, neither with regards to memory nor processing requirements. It is used very frequently in the SECO implementation, however, and a worthwhile improvement would be to build a highly optimised version of the array template, for example based on hashed lists.

5.2 SECO Components

As depicted in figure 5.1 on page 72, the SECO layer is composed of five components. The components are used to build two libraries and one executable which together constitute the SECO system.

`libcrep.a` is a library which provides dynamic linking support. It is built from the *class repository* code covered in section 5.2.1 and has to be linked with the application at compile time. This is a standalone library which can be used indepently of SECO.

`libseco.a` is a library which contains all the code required for SECO support. It is built from the *communications, constraint* and *event managers* covered in sections 5.2.2 to 5.2.4. This library must also be linked with the application at compile time.

`air` is the Application Instance Register (AIR) server executable. It is covered in section 5.2.5 and must be running on one node in the system before the application can be started.

The following sections describe the five SECO components, their APIs and any important details as to their working. Further details are available in the source code.

5.2.1 Class Repository

As mentioned, the class repository is essentially an extension of the CLASS REGISTER described in section 5.1.3. All save two of the source files in the directory were taken from [Fur97]. Changes were made to the original files in order to separate the CLASS REGISTER from the sample application it came with and turn it into a library. The new files are,

```
ClassRepository.hh
ClassRepository.cc
```

The remaining source files are documented in [Fur97] and will not be discussed in further detail in this thesis. When compiled, the files are used to create the library `libcrep.a` which must be linked into the application.

API

During its initialisation phase, an application instance creates a class repository, using the C++ `new` operator. Subsequent API invocations are performed on this object. The class repository contains two public methods, both of which are designed to be invoked by the application during its initialisation phase.

```
int Install(int language_id,                // implementation language
            int class_number,               // class number
            char * class_name,              // symbolic class name
            size_t size,                    // sizeof() the class
            int version,                    // class version number
            char * class_code,              // name of .so file
            char * upcall_code);            // name of .so file
```

The `Install` operation is used by the application to register any dynamic classes it may have. The operation is invoked once for each such class. The application is responsible for allocating and managing all of the above identifiers. The operation returns 0 if successful and -1 otherwise.

```
int Register(char * class_name,
             int version,
             int language_id);
```

Using the `Register` operation is not strictly necessary, since it implements a subset of the `Install` operation, but may be useful for some applications to reduce initialisation overhead. It registers a class that is known already to be in the shared file system. Since no copying of class code is done, it has a much lower overhead than `Install`. `Register` returns 0 if successful and -1 otherwise.

Shared File System

The code for dynamic classes is stored in a shared file system, the path of which is defined in `ClassRepository.hh` as,

```
#define SECO_SHARED_FILESYSTEM "/home/haahrm/seco/src/shared"
```

The names of the files in the shared file system are constructed with the private method `make_name`. The current naming scheme uses the language identifier, the class name and the class version to generate the class file name. If another naming scheme is desired, this method can be rewritten quite easily.

The class repository uses advisory locking, as described in `flock(2)`, to prevent files from being corrupted by different application instances attempting to register the same classes simultaneously. Unfortunately, the FreeBSD 2.0.5 implementation of advisory locks does not work across machine boundaries and it is therefore possible for a distributed application to corrupt files. There is no immediate solution to this, except for an operating system upgrade. Newer versions of Unix, such as Solaris 2.6, supports file locking across machine boundaries.

Improvements

The class repository is subject to some improvements, especially with regards to file locking. The code, though functionable in its current form, would also need a cleanup, but due to time constraints this has not been done.

5.2.2 Communications Manager

The communications manager manages network connections to other application instances. Each communications manager has a KANGA connection

endpoint on which it listens for messages from peers. Some of these messages will be subscribe, unsubscribe and raise requests which the communications manager forwards to its local event manager. Others will be control messages internal to the communicating communications managers.

API

The communications manager is used only by one other SECO component, namely the event manager. The constructor and destructor are defined as,

```
CommunicationsManager(EventManager* em, EventFactory* ef);
~CommunicationsManager();
```

When creating a communications manager object, the event manager passes it a pointer to itself. This pointer is later used by the communications manager to perform upcalls with requests received from the network. This is depicted with an upwards arrow in figure 5.1 on page 72. The event manager also passes the constructor a pointer to an event factory object which the communications manager can use to create event objects from marshalled event data received from the network. The communications manager's constructor and destructor also perform initialisation and shutdown of network connections. This will be described in more detail shortly.

There are three operations corresponding to the three ECO primitives. Each is used by the event manager to propagate subscriptions, unsubscriptions and raise requests to remote nodes.

```
void sendSubscription(EventType eventType,
                      EntityID entityID,
                      Handler handler,
                      NotifyConstraint* nc,
                      int receiverAID = -1);

void sendUnsubscription(EventType eventType,
                        EntityID entityID,
                        Handler handler);

void sendRaise(Event& event, Array<ApplID>& receivers);
```

The three methods marshal the relevant data and transmit it over the network using KANGA.

Multiple Threads

A communications manager object has two threads, a *listen thread* which continually listens for requests from peers, and a *worker thread* which processes these requests sequentially. The two threads exchange information through a semaphore and an array controlled by a mutex. The semaphore is used by the listen thread to signal the worker thread each time a new request is ready for processing.

Interaction with Other Application Instances

When a communications manager object is created, it goes through an initialisation phase where it makes its presence known to the other application instances. The constructor of a communications manager performs the following steps,

```
1. Obtain the address of the AIR by reading it
   from the shared file system.
2. Register with the AIR.
3. Obtain a list of peer endpoints from the AIR.
4. Register with each of the other peers in any order.
```

Analogously, the communications manager destructor performs the following steps,

```
1. Deregister with the AIR.
2. Deregister with the other peers in any order.
```

In figure 5.1 on page 72, communication with the AIR is depicted as a sideways dotted arrow. At any point between the constructor and destructor invocations, a table of peers is maintained, containing the endpoints of all other communications managers. This table is continually updated, by the listen thread as well as the worker thread, to reflect the current state of the world. Since both threads use the table, access to it is controlled by a mutex.

Message Formats

There are five types of messages sent by the communications manager to its peers. Two of them are used to register and deregister as described above and the remaining three correspond to the three ECO operations. The format of the messages are shown in figure 5.2 on page 79.

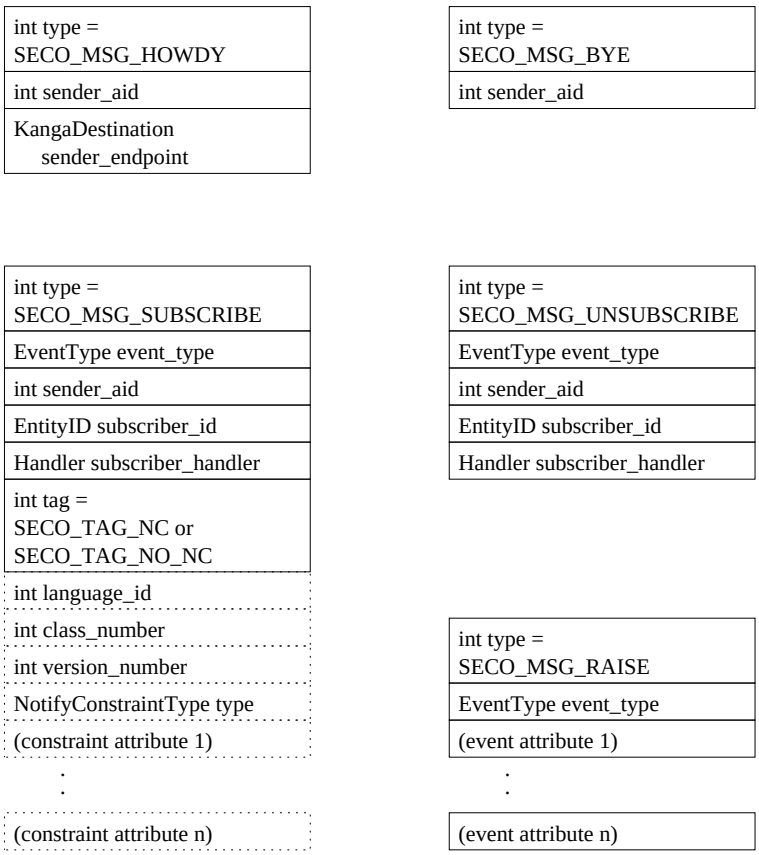

Figure 5.2: SECO Requests

It should be noted, that even though these requests are constructed by the communications manager, the marshalled attributes are added to the requests by the application. The communications manager explicitly invokes application code at the appropriate point when the request is constructed. All requests result in a reply of the forms shown in figure 5.3.

Figure 5.3: SECO Replies

Unicast and Multicast

The communications manager class implements two network communications primitives that are used internally (i.e., by the communications manager it-

self), *unicast* and *multicast*. Both methods are implemented using the underlying network programming package KANGA. Since multicast support in KANGA is still incomplete, the communications manager's multicast operation is implemented as a sequence of unicasts. When KANGA is extended to support multicast, the communications manager's multicast operation can easily be rewritten to use this feature.

5.2.3 Constraint Manager

The constraint manager manages notify constraints. It keeps a two-dimensional table of notify constraint information records, each containing information about a particular constraint object. Constraint objects can be in one of two states, either *linked* or *unlinked*. When the constraint manager evaluates a constraint, it first checks the state of the constraint. If unlinked, the constraint manager links the constraint before evaluation.

API

As shown on figure 5.1 on page 72, the constraint manager is only used by one other component, the event manager. The API is as follows,

```
ConstraintManager(NotifyConstraintFactory* cf);
~ConstraintManager();
```

When creating a constraint manager object, the event manager passes a notify constraint factory object to it. The constraint manager later uses this object to create notify constraint objects from marshalled constraint data. There are three operations for performing and cancelling subscriptions,

```
// for local subscriptions
virtual NotifyID RegisterNotifyConstraint(EventType et,
                                          NotifyConstraint& nc);
// for remote subscriptions
virtual NotifyID RegisterNotifyConstraint(EventType et,
                                          ClassIdentifier* ci,
                                          MarshalledData* mattribs);

// for local and remote unsubscriptions alike
void virtual DeregisterNotifyConstraint(EventType et, NotifyID nid);
```

The two registration methods register a linked and an unlinked notify constraint respectively. Notify constraints registered by a local entity are always linked, whereas notify constraints received over the network are assumed to

be unlinked.[1] Notify constraint class code is linked lazily and the objects are also unmarshalled lazily. This means that until an attempt to evaluate a constraint object is made, the constraint is kept in unmarshalled form. The two remaining methods are,

```
virtual Bool EvaluateNotifyConstraint(NotifyID&, Event&);

NotifyConstraint* LookupNotifyConstraint(EventType et, NotifyID nid);
```

The former is invoked by the event manager when it wants to match a particular event against a constraint. The latter when it needs access to a constraint object.

5.2.4 Event Manager

The event manager class is the application's API to the SECO library. It is the most complex of the five SECO components and since it binds three of the remaining four components together, it plays a central role in the implementation. This section describes the event manager implementation.

API

During its initialisation phase the application creates an event manager object. Subsequent ECO invocations (subscribe, unsubscribe and raise) are made on this object. The constructor and destructor are defined as,

```
EventManager(EventFactory* ef, NotifyConstraintFactory* cf);
~EventManager();
```

As parameters to the event manager constructor, the application passes two pointers to factory objects. These objects are needed by SECO to unmarshal events and notify constraints and instantiate real objects from the marshalled data in order to solve the type problem described in section 2.4.2. Examples of factory objects are given in section 6.2.2. The event manager class also contains three methods corresponding to the three ECO primitives that, after initialisation, can be invoked by the application. The declarations are,

[1] When a constraint object in marshalled form is received via the network, its class code can in reality be linked already if an instantiated object of the same class exists on that node. The constraint manager does not keep track of which classes are currently linked and always invokes the class repository. The class repository, however, knows what classes are linked and only attempts to link unlinked classes.

```
virtual void Subscribe(EventType et,         // event type
                       Entity* eptr,         // entity pointer
                       Handler handler,      // event handler pointer
                       NotifyConstraint* nc=NULL);

virtual void Unsubscribe(EventType et,       // event type
                         Entity* eptr,       // entity pointer
                         Handler handler);   // event handler pointer

virtual void Raise(Event& ev);
```

The following four methods are invoked by the communications manager. They are callback style methods and in figure 5.1 on page 72 they are depicted as a single upwards arrow.

```
virtual void iSubscribe(EventType et,              // event type
                        ApplID aid,                // subscriber's AI
                        EntityID eid,              // entity id valid at that AI
                        Handler handler,           // handler valid at that AI
                        MarshalledData *ma=NULL);  // optional marshalled nc

virtual void iUnsubscribe(EventType et,            // event type
                          ApplID aid,              // subscriber's AI
                          EntityID eid,            // entity id valid at that AI
                          Handler handler);        // handler valid at that AI

virtual void iRaise(Event& ev);                    // event

virtual void iResendSubscriptions(int aid);        // newly joined AI
```

The first three methods are ECO operations used for requests received over the network. The last method is used when the communications manager has learned of a new peer and wants the event manager to retransmit any active subscriptions it may have to this particular peer.

The Subscription Table

The event manager maintains a table of subscription records. Each record contains information about that particular subscription, e.g., whether the subscriber is local or remote, whether the subscription includes a notify constraint and which handler to invoke in case an event satisfying the constraint is received. The table is indexed in four dimensions by *event type, application instance identifier, entity identifier,* and a *serial number.* For a particular subscription, the indices constitute a large portion of the subscription information. The advantage of using a four-dimensional table (as opposed to, e.g., a single linked list) is that information which for most operations is

available anyway (such as the event type which is a parameter to all three ECO operations) can be used to limit the search space in a manner similar to hashed lists.

Multi-threading

As opposed to the communications manager, the event manager does not have its own threads. When invoked by the application, the application thread executes event manager code, meaning that SECO operations are synchronous. However, since the communications manager can make upcalls to the event manager, multiple threads can still be executing event manager code simultaneously. Therefore, access to the subscription table is controlled by a mutex.

Improvements

The event manager, the largest of the five SECO components, relies heavily on the subscription table which is implemented as a four-dimensional `Array` of the type described in section 5.1.4. A worthwhile improvement could therefore be to optimise the `Array` type.

5.2.5 Application Instance Register

The application instance register (AIR) maintains authoritative information about the application instances in the world at any time. It is a small standalone executable which must run on exactly one node in the system before any application instances can be started. There are no requirements as to which node. The AIR uses KANGA for communication with application instances and maintains information about application instances in the form of KANGA endpoints.

API

As described in section 5.2.2 new application instances use the AIR during their initialisation phase to register their presence and to obtain a list of their peers, and during their shutdown phase to deregister their presence. The AIR API used to perform these operations is different from the APIs of other SECO components in that it is not invoked directly but through a custom RPC mechanism. There are three AIR operations and for each there is a request and a response message.

`Register` is invoked by a new application instance during its initialisation phase. The registering AI passes its endpoint which goes into the AIR database. The AIR allocates a unique application instance identifier (an integer) and returns it to the application instance.

`List` is also invoked by a new application instance during its initialisation phase. After having registered, the new AI obtains a list of all AIs currently registered, including itself.

`Remove` is invoked by an application instance during its shutdown phase. The AIR removes it from its records.

In order to invoke the AIR operations, a new application instance must obtain the endpoint of the AIR. For this reason, the AIR writes the address of its endpoint into a file in the shared file system during its initialisation phase. New application instances know the name of the file (it is set at compile time) and read it before attempting to invoke the AIR.

Message Formats

The three API operations are implemented as three request/response dialogues. They are shown in figure 5.4.

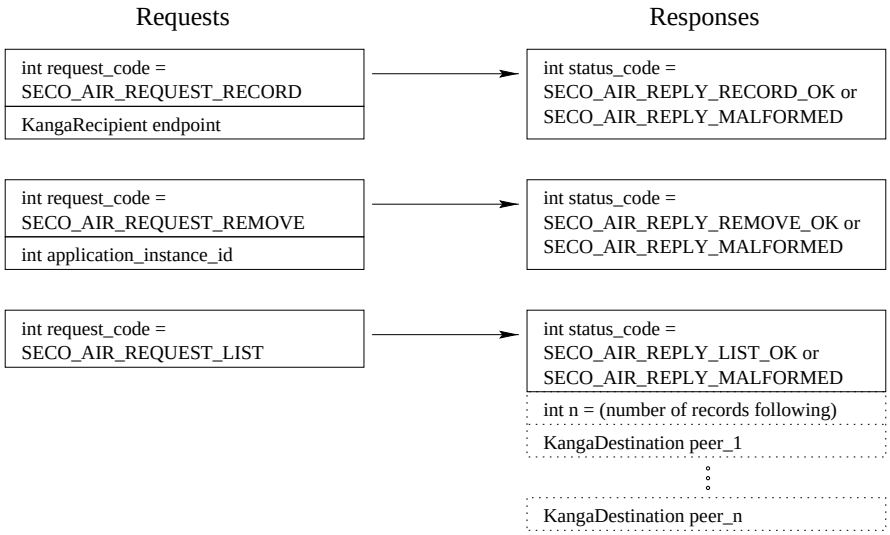

Figure 5.4: AIR Messages

Consistency

During initialisation, an application instance receives authoritative information from the AIR about the state of the world. After that the AI is solely responsible for maintaining its own view, meaning that it does not query the AIR again but only updates its records according to messages received from other AIs. Changes are propagated eagerly during normal circumstances (e.g., a new AI sends a *bye* type message to all peers during shutdown) but lazily in case of failures. Hence, when an application instance discovers that a peer has failed, no attempt is made to notify other AIs or the AIR. They are assumed to discover failures on their own if they need to know about them. This is a fairly lax consistency model that scales reasonably well.

Improvements

Since the AIR is not only vital to the operation of SECO but also centralised, it is a single point of failure. Though it is a very simple program (consisting of less than 300 lines of source code) and therefore less likely to contain bugs than a complex program, it is still subject to hardware, operating system and network failures. An improvement would be to implement a distributed version of the AIR, making it less prone to such failures.

As mentioned, discovery of failures is not propagated from application instances to the AIR. Since the AIR never initiates communication, this means that it may never discover if an application instance has failed without deregistering. In the current implementation, this is not a problem since KANGA connections between the AIR and the application instances are kept open and the AIR will therefore discover if an AI fails. However, keeping connections open may not be a viable solution in the long run and an improvement could be to devise a better scheme.

5.3 Summary

This chapter has described the distributed ECO implementation. We have outlined the overall structure of the system by separating it into three conceptually distinct levels, each holding a number of system components. Each component has been described in detail and the relationship between them has been explained. Since the application layer is not really part of SECO, description of the sample application has been delayed until section 6.2.

Chapter 6

The Active Badge System

This thesis relies to a great extent on an event-based system found at the University of Cambridge. The system, called an *active badge system*, keeps track of people in research laboratories by raising so-called *sighting* events. The experiments presented in chapter 7, and the subsequent evaluation of scalability in the ECO model, are based on event data obtained from the Cambridge system.

This chapter describes the active badge system in detail. First, section 6.1 takes a look at the physical system found at the University of Cambridge and the nature of the event data we have from it. Second, section 6.2 describes the simulation used in chapter 7 in implementation-level detail, and finally, section 6.3 sums up. The simulation also serves as an example of an application using SECO support.

6.1 Badge System Description

The active badge system[1] consists of a collection of infrared sensors (called *stations*) which pick up signals emitted by battery-driven badges worn by personnel in some university laboratories. The stations are grouped into *networks*, each being a segment of a particular laboratory. In addition, users can also be detected when they log into the campus computer network, e.g., via an X-terminal. Each badge carries a unique *badge identifier*, a six byte value which is picked up by the sensors. Certain kinds of equipment, such as workstations, X-terminals and some network devices, also have a badge identifier associated with them and can cause sighting events to be raised.

[1]Described in `http://www.cl.cam.ac.uk/abadge/documentation/abinfo.html`.

The Data

The data used in these experiments consists of 1.8 megabytes of real data sampled from the active badge system at the Olivetti/Oracle Research Lab at Cambridge University. The data consists of 35,811 badge sightings collected over a period of almost 21 hours by 118 stations distributed over 12 networks. For each sighting, we have the following information,

Station Identifier identifying the network (by a symbolic name) and the station (by an integer) within that network.

Badge Identifier identifying the sighted person or equipment (by a sequence of six eight-bit hexadecimal numbers separated by dashes).

Time stamp identifying the moment when the sighting was made in seconds and microseconds, since 00:00:00 UTC, January 1, 1970.[2]

The experimental strategy is to replay these sightings in a simulation in form of a series of programs using SECO support. We represent each station as an ECO entity which raises the sighting events recorded in the Cambridge data at the appropriate times, as measured by the local system clock. The simulation is described in section 6.2 and the experimental configuration in chapter 7.

6.2 Simulation Implementation

This section describes the implementation of the active badge system simulation. The simulation serves two purposes. First, it is used in the experiments in chapter 7. Second, it is used as a demonstration of how SECO is used by an actual application. The simulation uses data gathered from the real badge system to replay actual events which happened during a 21 hour period in January 1998 at the University of Cambridge. The reader is expected to be familiar with the CREP and SECO APIs given in section 5.2.1 and 5.2.4 respectively.

6.2.1 ECO Classes

This section describes the entities, events and constraints used in the badge system. Each is implemented as a C++ class derived from a SECO abstract base class. Notify constraint classes are more complex than entity and event classes because they need support for dynamic linking.

[2] As returned by `time(3)`.

Entities

The entities used in the badge system can be grouped in two, *raising* and *receiving* entities. Though it is perfectly possible for ECO entities to raise as well as receive events, the nature of the badge system is such that its entities do either but not both. Entities are derived from the `Entity` abstract base class.

The entity source code is found in the files,

```
Entities.hh
Entities.cc
```

There are a total of five types of entities, each implemented as a C++ class. Each class uses a notify constraint object to filter events. The five entity classes are,

`StationEntity` corresponds to an infrared sensor in the real badge system and is the only type of entity that can raise events. Whenever it sights a particular badge, it raises a *Sighting* event which includes the unique badge identifier of the sighted badge. Like the physical sensors, each station entity is identified by a network (e.g., `ORL-Three#8`) and a station number (an integer) valid within that network. They are passed as parameters to the constructor when the station entity is created. Each station entity has a thread which reads through a log file from the real badge system and raises events at the appropriate times.

`GodEntity` subscribes to all events generated by all stations. Hence, the God entity sees all.

`SecurityEntity` emulates a closed circuit television (CCTV) security camera. It records all events generated in a particular network. (A network is usually covered by multiple sensors.)

`PrivateEntity` emulates a private detective shadowing a particular badge. This entity receives all sighting events generated by the badge in question, no matter where they originate from.

`BrotherEntity` emulates *Big Brother* from [Orw90]. Big Brother has three modes and can survey either sightings generated by human users, electronic hardware,[3] or by unlisted badges. Three different big brother entities see as much as one God entity.

[3]Badge identifiers can be given to certain kinds of hardware such as a printer or a workstation.

Events

Only one type of event is used in the badge system. It is the sighting event which is raised by a sensor when a particular badge is sighted. Events are derived from the `Event` abstract base class. The source code for the sighting event can be found in the files,

```
Events.hh
Events.cc
```

A sighting event has the form,

```
SightingEvent(char *network,    // name of the network
              int station,      // the station within that network
              char *badge,      // the unique id of the badge seen
              long time_sec,    // when it was seen (seconds)
              long time_usec);  // when it was seen (microseconds)
```

Constraints

There are four types of notify constraints, each corresponding to one of the subscribing entities mentioned above. Notify constraint classes are derived from the `NotifyConstraint` abstract base class. Since constraints are dynamically linkable, their source code is more complicated than that of entities and events. For example, the big brother entity mentioned above uses a notify constraint called `BrotherNC`, the source code of which is stored in the following files,

`BrotherNC.hh` is a header file which defines the interface of the class.

`BrotherNC.cc` contains stub code that is linked with the application at compile time.

`BrotherNC.C` contains the real class implementation which, if needed, is linked dynamically with the application at runtime.

`BrotherNC_Upcall.C` contains upcall code required for dynamic linking.

Ideally, the DCLASS preprocessor should be used to generate the stub and upcall code but, as mentioned in section 5.1.3, it is not stable enough.

6.2.2 Additional ECO Support

This section describes the additional support included in the badge system application which is required to use the SECO library.

Factory Objects

The application code must include two factory classes which can be used to create event and constraint objects respectively. These factory objects are used to solve the type problem described in section 2.4.2 by mapping type identifiers to instantiated objects. An example of such a mapping function is,

```
// from Events.cc
Event* BadgeSystemEventFactory::Create(EventType et, MarshalledData& md) {
  switch(et) {
  case BADGE_EVENT_Sighting:
     return(new SightingEvent(md));
  ...
  }
}
```

If the application contains many types of events and constraints, this `switch` statement can obviously grow rather big. However, the code is simple and an obvious candidate for automatic generation.

Initialisation and Shutdown

The application must also include code to initialise the various parts of the SECO system. Initialisation code for the badge system is,

```
// Initialise Roo
Roo::init();

// Create class repository and install dynamic classes
ClassRepository* grt = new ClassRepository();
grt->Install(CPP, BROTHER_NC, "BrotherNC", sizeof(BrotherNC), 0,
             "BrotherNC.so", "BrotherNC_Upcall.so"))
// (repeat Install for each dynamic class)

// Create factories and event manager
BadgeSystemEventFactory* ef = new BadgeSystemEventFactory();
BadgeSystemNotifyConstraintFactory* cf = new BadgeSystemNotifyConstraintFactory();
EventManager* em = new EventManager(ef, cf);
```

At this stage, the SECO system is ready and the application can start creating entities and let them perform subscriptions. When the application wants to shut down, it should execute the following code,

```
delete grt;
delete em;
Roo::shutdown();
```

Additional Support for Dynamic Linking

In addition to the four source files for each dynamically linkable class listed in section 6.2.1, the application should allocate identifiers used to register dynamically linkable classes. These identifiers are passed as parameters to the class repository's `Install` method. The declarations used in the badge system are,

```
// from BadgeSystem/Dynamic.hh
enum LanguageIdentifier { CPP };
enum DynamicClasses { GOD_NC, SECURITY_NC, PRIVATE_NC, BROTHER_NC };
#include "crep/Dynamic.hh"
```

The first declaration enumerates the possible implementation languages. C++ is the only one currently used. The second enumerates all the dynamic classes known to the system.

6.3 Summary

This chapter has described the active badge system and our simulation of it. The simulation was used to show what an application using SECO support can look like, and will be used in the next chapter to evaluate scalability in the ECO model.

Chapter 7

Experiments and Evaluation

This chapter presents some experiments designed to evaluate whether the implementation fulfills the three objectives listed in section 1.6. The application used is the active badge system simulation described in chapter 6, and real data, gathered from the Cambridge system, is used as input to it. The simulation was one of several candidates for an experimental application (another was the tank game mentioned in chapter 2) but the badge system simulation was chosen primarily because of its physical counterpart being a very 'real' system. First, the events used in the simulation were generated by real users going about their daily tasks in the hallways of a real college. Furthermore, the active badge system has no less than 118 stations which means it is a rather big system. Size is a common characteristic of event-based systems used in the industry, but one which is difficult to simulate in a research environment. These characteristics made the active badge system highly attractive as an experimental application, since any results obtained with it would have a high degree of practical relevance.

The experiments presented in this chapter show filters to be highly effective and certainly worthwhile as a means to decrease network traffic. These findings, documented in section 7.3, are extremely positive. On the other hand, dynamic linking is shown to have only a minor effect on application footprint and at a fairly high cost. These findings, described in section 7.4, are less positive. The experiments also show that the SECO implementation works well and is perfectly usable as a research environment.

To understand the experiments in sections 7.3 and 7.4, this chapter begins with two other sections. The first makes a few clarifications to the three objectives with regards to measurement technique. The second, section 7.2, gives general background information used in the rest of the chapter. Most importantly, it describes the configuration of the simulation and presents an analysis of the administrative overhead inherent in it. The results of this

analysis are used in the subsequent treatment of the experiments, and the analysis is therefore presented before the experiments themselves.

7.1 Clarifying the Objectives

This section discusses and clarifies certain parts of the objectives presented in section 1.6, in particular with respect to measuring the various application characteristics.

7.1.1 Measuring Network Traffic

Objective C (section 1.6.3) said,

> *"One of our goals is to measure the number of network messages saved by the use of [...] filtering [...]"*

This phrasing specifies that gain due to filtering should be measured at the level of network messages. Since the communications package we are using (KANGA) is message oriented, this is fairly straightforward to implement. Messages do vary in size, however, and another approach could be to measure the number of bytes transmitted across the network. Had there been a SECO-induced per message overhead,[1] measuring the number of bytes would certainly have been relevant, since there would then have been a relationship between notify constraints and message sizes. However, in SECO message sizes vary only according to the number of (application-defined) event parameters, and we therefore uphold the decision to measure number of messages rather than the exact amount of bandwidth used.

7.1.2 Measuring Code Complexity

Objective B (section 1.6.2) said,

> *"We want to determine how much extra complexity (in terms of code) is needed from the application to use this facility [...]"*

Measuring the complexity of a given line of source code is a very subjective process and one that is hardly possible to perform generally in any meaningful manner. In this chapter we therefore measure complexity in terms

[1] For example, if a raise message contained the identifiers of all receiving entities at the receiving side. As explained in section 4.1.2, this would eliminate the need to evaluate notify constraints at the receiving side.

of additional number of lines or source code required to support dynamic linking, regardless of whether the code is easy or difficult for an application programmer to read or write.

7.1.3 Measuring Footprint

Objective B also said,

> *"[...] and to measure how large the gain is in terms of decreased application footprint."*

Measuring footprint can be done simply by looking at the sizes of binary (i.e., compiled) files and is therefore fairly trivial. However, footprint may vary according to various factors, such as the underlying hardware platform, operating system, compiler and debugging information included by the compiler. In this chapter, we measure footprint only for one combination of such factors, as described in section 7.2.1.

7.2 Background

This section provides background information that is necessary for understanding sections 7.3 and 7.4, which describe the actual experiments performed. Section 7.2.1 describes the platform on which the experiments are performed. Section 7.2.2 describes general characteristics of the badge system simulation and section 7.2.3 analyses these characteristics in the context of network traffic overhead caused by SECO.

7.2.1 Hardware, OS, and Compiler Configuration

Our testbed consists of four 80486/DX2 (33/66 MHz) based PCs with 16 megabytes of memory, running FreeBSD 2.0.5. The machines are connected with standard 10 Mbps ethernet. Since we are measuring bandwidth usage on a per message basis (as opposed to, e.g., roundtrip times) we run the experiments in multiuser mode. Also, the machines are on a network segment with traffic not related to the experiments. The only available compiler, GCC 2.6.3, is used to compile the programs and standard debugging information is generated with the `-g` flag. The binary code is not stripped.[2]

[2] As with the `strip(1)` command.

7.2.2 Simulation Configuration

In our simulation, each of the twelve networks in the badge system is represented by one application instance as shown in figure 7.1 on page 96. Each instance holds an entity for each station in that network as shown in the figure. Three of the four nodes (`janis, zoot` and `hoghtrob`) are used to hold these application instances. This configuration is the same for all the experiments in this chapter. The last node (`statler`) holds the application instance register (AIR) and a thirteenth application instance with the entities that subscribes to events. The actual subscribers held by this application instance varies between experiments. This configuration has three characteristics,

1. All subscribing entities are held by a single application instance.
2. All event-generating entities are held by other application instances.
3. All application instances holding event-generating entities outlive those with subscribing entities.

These characteristics are used in the discussion in section 7.2.3 where we examine the administration overhead involved for an application with these characteristics.

Entity Distribution

Though we know the distribution of stations within networks in the original badge system, we do not have details as to their interconnection. For example, figure 7.1 shows all stations within a network to be local to each other, but whether this was also true for the original system is unknown. However, the modelling of the system only has to be accurate enough to assure that the results obtained are valid. Since the objective is to measure network bandwidth, it is sufficient that events are sent (or would have been sent, but are eliminated by a filter) over the network. In SECO, this means that the raiser and the subscriber must reside in different application instances. In case there are multiple raisers, like the 118 stations in figure 7.1, their exact distribution is unimportant, as long as any subscribers are remote, i.e., do not reside in the same application instances.

The simulation takes place in real time (taking 22 hours to raise all events) and there are 178 subscribers in total. Therefore, it is not possible to run all subscribers separately within a reasonable time frame. Instead, most of the experiments features multiple subscribers running in parallel on the same application instance. This has no influence on the measurements.

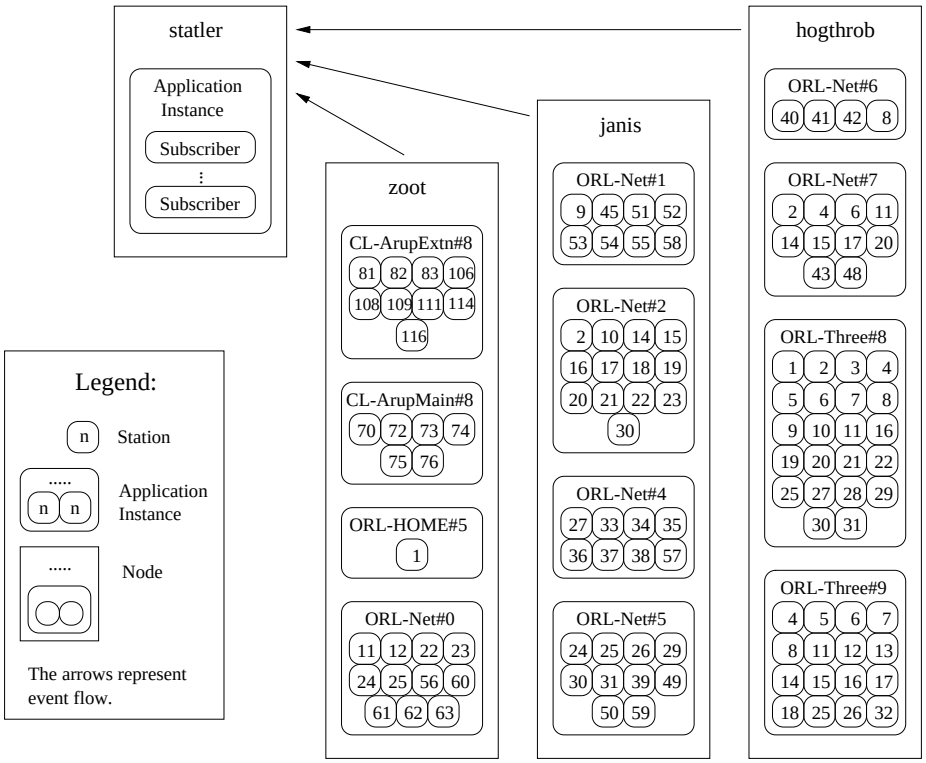

Figure 7.1: Badge System Simulation Overview

Timing Considerations

The event data that we have available contains very precise timestamps (seconds and microseconds) and an accurate simulation would have raised events at these instants very precisely. However, with the system support available on FreeBSD, it is not possible to raise events at such accurate moments. Furthermore, each node has its own clock and since the entities raising the events reside on different nodes, it would also be necessary to continually resynchronise the system clocks on the different nodes. The lack of a global clock is a well-known problem from distributed systems theory.

However, raising the events at exactly the moments they were measured in the original badge system is not important. As discussed in section 4.2.2, the events supported by SECO are not strictly ordered. Hence, even if the simulation could raise them at the right moments, SECO could not guarantee the delivery order. Instead, the original time stamps are given as parameters to the sighting events, allowing subscribers interested in the order of the events to compare these time stamps rather than examine the order in which

the events are received.[3]

To avoid excessive clock drift, the four nodes are running the network time protocol (*ntp*) which keeps the system clocks on the four nodes synchronised to that of an external time server.

7.2.3 Administration Overhead

Like any distributed system, the SECO implementation introduces a certain overhead, in terms of network traffic, which would not have applied in an equivalent centralised solution. This section analyses the configuration described in section 7.2.2 and provides details as to the size of this overhead. The discussion is based on our knowledge of the implementation and the results will be used later in section 7.3. The analysis applies to any scenario subject to the three characteristics listed in section 7.2.2. The discussion in this section does not apply for applications that do not fulfill the three characteristics but of course apply for the four test scenarios presented in section 7.3.

Administration overhead falls in two groups: overhead caused by *distribution* and by *filtering*. The former consists of overhead caused by AIR interaction and by maintaining group memberships, whereas the latter is caused by subscription and unsubscription messages. In the following, we compute the overhead of each category as a function of the number of application instances (rather than entities) and subscriptions. We define the number of overhead messages sent due to distribution and filtering, respectively, as,

$$\begin{aligned} N_{distribution}(n) &= N_{air}(n) + N_{group}(n) \\ N_{filtering}(m,n) &= m \times N_{subscription}(n) \end{aligned}$$

where n is the number of application instances and m the number of subscriptions performed during the application's lifetime.

AIR Interaction Overhead

During its lifetime, an application instance sends three requests to the AIR: *register, list* and *deregister*. Each request results in a reply. This makes a total of six messages per application instance. For a scenario with n application instances, AIR interaction therefore results in a total of $6n$ messages being transmitted across the network during the application's lifetime, thus,

$$N_{air}(n) = 6n$$

[3]Event order is not used in the simulation experiments but could, for example, be used to implement event composition as known from the Cambridge event model described in section 2.3.

Group Administration

When an application instance joins and leaves a scenario, respectively, it multicasts *howdy* and *bye* messages to all the other application instances, a list of which it received from the AIR. The *howdy/bye* messages are effectively used to maintain membership of a process group including all application instances. *Howdy* is used to join the group and *bye* to leave it. Since the current SECO implementation does not use multicast to transmit these messages, they are sent as a sequence of unicasts. This is unfortunately highly inefficient. An application instance which joins a scenario as the nth application instance needs to send $n-1$ howdy messages. This means that in a scenario with n application instances (where none have left) the total number of *howdy* messages sent will be,

$$\sum_{i=1}^{n-1} i = n \times \frac{n-1}{2} = \frac{n^2 - n}{2}$$

Since each application instance will also have to send a corresponding *bye* message, the total number of *howdy/bye* messages is,

$$2 \times \frac{n^2 - n}{2} = n^2 - n$$

For each message, there will also be a reply,[4] meaning that the total number of messages caused by group administration is,

$$N_{group}(n) = 2(n^2 - n)$$

Note that these unicast messages could have been replaced by only $2 \times (n-1)$ multicast messages, had multicast been available.

Subscription Overhead

In the scenarios used in this chapter, all subscribing entities are hosted by a single application instance (assumptions 1 and 2 from section 7.2.2). Hence, since subscriptions are broadcast to all other instances, the total number of messages transmitted over the network for any particular subscription can be computed easily. If n_s is the number of application instances at subscription time, the number of subscription messages (and replies) sent is,

$$2(n_s - 1)$$

[4] KANGA requires that a reply be sent to a unicast message.

At unsubscription time, the number of application instances may have changed. Assuming it is called n_u and that there is an unsubscription for each subscription, the total number of subscribe/unsubscribe messages is,

$$2(n_s - 1) + 2(n_u - 1)$$

In the scenario used in our experiments, the application instance with event-generating entities always outlive the one with the subscriber (assumption 3 from section 7.2.2), so for this particular case we have $n_s = n_u$. Setting $n = n_s = n_u$ the total number of subscribe/unsubscribe messages is,

$$N_{subscription}(n) = 2(n_s - 1) + 2(n_u - 1) = 4(n - 1)$$

Note that these unicast messages could be replaced by only 2 multicast messages, had multicast been available.

Total Overhead

The total overhead caused by distribution can now be computed as,

$$\begin{aligned} N_{distribution}(n) &= N_{air}(n) + N_{group}(n) \\ &= 6n + 2(n^2 - n) \\ &= 2n^2 + 4n \end{aligned}$$

And for filtering as,

$$\begin{aligned} N_{filtering}(m, n) &= m \times N_{subscription}(n) \\ &= m \times 4(n - 1) \\ &= 4m(n - 1) \end{aligned}$$

7.3 Event Filtering Experiments

This section addresses *Objectives A* (to provide a distrbuted implementation) and *C* (to measure filtering gain) by presenting and discussing results from a series of experiments run on a distributed application using SECO support. In particular, the results are used to determine the cost and benefit of using notify constraints to filter events. We used the configuration described in section 7.2.2 (varying the subscribers between experiments) to measure the number of messages transmitted, and the formulas from section 7.2.3 to calculate administration overhead.

Choosing Scenarios

In practice, the complete event flow through an active badge system is large and difficult to comprehend. Subscribers with well-chosen notify constraints can be used to provide meaningful views of this event flow by dynamically extracting events according to certain patterns, and in this way make it easier for humans to monitor the system at runtime. The four subscribing entities presented in this chapter were designed to present such meaningful views of the event flow and would be likely candidates for implementation in a real (non-simulation) badge system. The test scenarios are covered in sections 7.3.1 to 7.3.4. They involve entities subscribing to different sets of events, in the manner described below,

Subscriber 1: God sees all and gets all events. In a real badge system, this subscriber could be useful for logging purposes. In this chapter it is also used to measure filtering overhead by implementing a filter without effect. Covered in section 7.3.1.

Subscriber 2: Security Camera subscribes to all events generated in a particular network. In a real system, it could be used to monitor a specific (and therefore more manageable) area of the entire system. This experiment was run with twelve subscribers in parallel, one for each network. Covered in section 7.3.2.

Subscriber 3: Private Eye subscribes to all events generated by a particular badge. In a real system, this subscriber could be used to trace the movement patterns of a particular badge owner. This experiment was run with 162 subscribers in parallel, one for each badge present in the event data. Covered in section 7.3.3.

Subscriber 4: Big Brother has three modes: it subscribes either to events generated either by users, by equipment or by unlisted badges (badges for which the owner's type is unknown). In a real system, it could be useful for logging purposes. This experiment was run with an entity of each type in parallel. Covered in section 7.3.4.

Computing Filtering Overhead

Since the configuration used is the same for all experiments, the number of overhead messages caused by administration does not vary and can be computed in advance. However, we are measuring the costs and benefits of *filtering* (as opposed to distribution) and are therefore only interested in the filtering overhead.

Entity	*Unfiltered event msgs*	*Actual event msgs*	*Overhead msgs*	*Total msgs*	*Relative Decrease*
God	71,622	71,622	48	71,670	−0.1%

Table 7.1: Results from Experiment 1

In each experiment, there are thirteen application instances, as described in section 7.2.2, so $n = 13$. In each experiment,[5] one subscription is made, so $m = 1$. Inserting in one of the formulas from section 7.2.3 gives us,

$$\begin{aligned} N_{filtering}(m, n) &= 4m(n-1) \\ N_{filtering}(1, 13) &= 4{\times}1{\times}(13-1) \\ &= 48 \end{aligned}$$

This means that in each experiment, a total of 48 messages will be sent in order to support filtering.

7.3.1 Experiment 1

The `GodEntity` used in this experiment subscribes to all events generated in the badge system by implementing a notify constraint that returns true regardless of which sighting event it is matched against. Though without effect, the notify constraint is still propagated during subscription, causing a slight overhead.

Table 7.1 shows the results from the experiment. As can be seen, the reduction in number of messages is negative, meaning that using the worthless filter (not suprisingly) introduced some overhead. However, in the experiment it was as low as 0.1%. It is important to look at the scenario in which this result was obtained. Overhead in the form of extra network messages is generated at subscription and unsubscription time but not while the subscription is in effect. In experiment 1, there was only one subscription involved and it was in effect for a very long time (time enough to raise 35,811 events). Consequently, the relative cost decreased as more and more bandwidth was used for other purposes. We conclude that long-lasting subscriptions have a relatively low overhead.

7.3.2 Experiment 2

The subscriber used in this experiment is a simulation of a CCTV (closed circuit television) camera which monitors a specific area. As described in

[5] In experiments 2 to 4 where multiple subscribers were run in parallel, we count each subscriber as a separate experiment.

Security camera	*Unfiltered event msgs*	*Actual event msgs*	*Overhead msgs*	*Total msgs*	*Relative Decrease*
CL-ArupExtn#8	71,622	104	48	152	99.9%
CL-ArupMain#8	71,622	44	48	92	99.9%
ORL-Home#5	71,622	14	48	62	99.9%
ORL-Net#0	71,622	4,250	48	4,298	94.0%
ORL-Net#1	71,622	11,406	48	11,454	84.0%
ORL-Net#2	71,622	13,864	48	13,912	80.6%
ORL-Net#4	71,622	5,164	48	5,212	92.7%
ORL-Net#5	71,622	10,046	48	10,094	85.9%
ORL-Net#6	71,622	4,150	48	4,198	94.1%
ORL-Net#7	71,622	3,798	48	3,846	94.6%
ORL-Three#8	71,622	14,288	48	14,336	80.0%
ORL-Three#9	71,622	4,494	48	4,542	93.7%
Average					91.6%

Table 7.2: Results from Experiment 2

section 7.2.2 there are twelve networks in the active badge system. In this experiment, twelve security camera entities were run in parallel, each subscribing to all events generated in a particular network. The result is shown in table 7.2.

As can be seen, the reduction in number of transmitted messages is quite high: above 90% on average. Had events simply been broadcast instead of filtered, approximately ten times as many messages would have been transmitted across the network. Even the most busy camera only received 20% of the messages it would have received if filters had not been used. As in experiment 1, these subscriptions were in effect for a long period of time, and the fixed administration overhead of 48 messages gradually became less and less significant as more events were raised.

7.3.3 Experiment 3

The subscriber used in this experiment is nicknamed *private eye* because of its likeness to a private detective shadowing a particular person. In this experiment, 162 private eye entities were run in parallel, one for each badge present in the event data. Each entity used a notify constraint to subscribe to events generated by one specific badge regardless of its whereabouts. Tables 7.3 to 7.5 (on page 103, 104 and 105 respectively) show the results from the experiment.

The data in the three tables shows substantial savings for this scenario, averaging at around 99.2% reduction in the number of messages transmitted across the network. The private eye entities in this experiment collectively get all sightings of registered badges. The busiest of the 162 badge-wearers caused 1,316 sightings and still received only 2% of the messages it would have received if filters had not been used.

badge watched	*unfiltered event msgs*	*actual event msgs*	*overhead msgs*	*total msgs*	*relative decrease*
0-0-0-0-10-14	71,622	52	48	100	99.8%
0-0-0-0-10-1a	71,622	16	48	64	99.8%
0-0-0-0-10-1b	71,622	1,202	48	1,250	98.2%
0-0-0-0-10-1c	71,622	1,302	48	1,350	98.0%
0-0-0-0-10-1e	71,622	48	48	96	99.8%
0-0-0-0-10-27	71,622	16	48	64	99.8%
0-0-0-0-10-2a	71,622	74	48	122	99.7%
0-0-0-0-10-31	71,622	134	48	182	99.6%
0-0-0-0-10-34	71,622	78	48	126	99.7%
0-0-0-0-10-39	71,622	60	48	108	99.7%
0-0-0-0-10-4	71,622	784	48	832	98.7%
0-0-0-0-10-9	71,622	52	48	100	99.8%
0-0-0-0-12-11	71,622	82	48	130	99.7%
0-0-0-0-12-21	71,622	322	48	370	99.4%
0-0-0-0-12-26	71,622	66	48	114	99.7%
0-0-0-0-12-28	71,622	292	48	340	99.4%
0-0-0-0-12-2a	71,622	22	48	70	99.8%
0-0-0-0-12-34	71,622	36	48	84	99.8%
0-0-0-0-12-35	71,622	294	48	342	99.4%
0-0-0-0-12-36	71,622	90	48	138	99.7%
0-0-0-0-12-4	71,622	512	48	560	99.1%
0-0-0-0-12-67	71,622	8	48	56	99.8%
0-0-0-0-12-9	71,622	106	48	154	99.7%
0-0-0-0-12-d4	71,622	4	48	52	99.8%
0-0-0-0-13-64	71,622	18	48	66	99.8%
0-0-0-0-13-99	71,622	4	48	52	99.8%
0-0-0-0-13-9a	71,622	400	48	448	99.3%
0-0-0-0-13-e7	71,622	272	48	320	99.5%
0-0-0-0-13-e8	71,622	408	48	456	99.3%
0-0-0-0-13-e9	71,622	1,316	48	1,364	98.0%
0-0-0-0-13-ea	71,622	590	48	638	99.0%
0-0-0-0-14-0	71,622	748	48	796	98.8%
0-0-0-0-14-4	71,622	478	48	526	99.2%
0-0-0-0-14-5	71,622	4	48	52	99.8%
0-0-0-0-1b-d6	71,622	4	48	52	99.8%
0-0-0-0-4-0	71,622	62	48	110	99.7%
0-0-0-0-4-1	71,622	742	48	790	98.8%
0-0-0-0-4-11	71,622	640	48	688	98.9%
0-0-0-0-4-1f	71,622	70	48	118	99.7%
0-0-0-0-4-2b	71,622	24	48	72	99.8%
0-0-0-0-4-2f	71,622	732	48	780	98.8%
0-0-0-0-4-4	71,622	450	48	498	99.2%
0-0-0-0-4-44	71,622	402	48	450	99.3%
0-0-0-0-4-74	71,622	72	48	120	99.7%
0-0-0-0-4-8	71,622	508	48	556	99.1%
0-0-0-0-4-a2	71,622	632	48	680	99.0%
0-0-0-0-5-78	71,622	462	48	510	99.2%
0-0-0-0-6-bd	71,622	360	48	408	99.3%
0-0-0-0-7-10	71,622	240	48	288	99.5%
0-0-0-0-7-2	71,622	200	48	248	99.6%
0-0-0-0-7-8	71,622	318	48	366	99.4%
0-0-0-0-79-0	71,622	4	48	52	99.8%
0-0-0-0-80-29	71,622	710	48	758	98.8%
0-0-0-0-80-91	71,622	624	48	672	99.0%

Table 7.3: Results from Experiment 3, Part 1 of 3

badge watched	*unfiltered event msgs*	*actual event msgs*	*overhead msgs*	*total msgs*	*relative decrease*
0-0-0-0-80-ad	71,622	666	48	714	98.9%
0-0-0-0-81-1e	71,622	640	48	688	98.9%
0-0-0-0-81-21	71,622	706	48	754	98.8%
0-0-0-0-81-4c	71,622	638	48	686	98.9%
0-0-0-0-81-51	71,622	388	48	436	99.3%
0-0-0-0-81-55	71,622	670	48	718	98.9%
0-0-0-0-81-5a	71,622	702	48	750	98.9%
0-0-0-0-81-5b	71,622	718	48	766	98.8%
0-0-0-0-81-64	71,622	640	48	688	98.9%
0-0-0-0-81-67	71,622	20	48	68	99.8%
0-0-0-0-81-6f	71,622	686	48	734	98.9%
0-0-0-0-81-7c	71,622	208	48	256	99.5%
0-0-0-0-81-7e	71,622	596	48	644	99.0%
0-0-0-0-81-83	71,622	220	48	268	99.5%
0-0-0-0-81-95	71,622	640	48	688	98.9%
0-0-0-0-81-98	71,622	686	48	734	98.9%
0-0-0-0-81-9d	71,622	290	48	338	99.4%
0-0-0-0-81-a2	71,622	692	48	740	98.9%
0-0-0-0-81-ab	71,622	668	48	716	98.9%
0-0-0-0-81-ad	71,622	588	48	636	99.0%
0-0-0-0-81-b4	71,622	626	48	674	99.0%
0-0-0-0-81-ba	71,622	672	48	720	98.9%
0-0-0-0-81-cd	71,622	676	48	724	98.9%
0-0-0-0-81-ce	71,622	644	48	692	98.9%
0-0-0-0-81-d3	71,622	600	48	648	99.0%
0-0-0-0-81-da	71,622	676	48	724	98.9%
0-0-0-0-81-dd	71,622	32	48	80	99.8%
0-0-0-0-81-e3	71,622	682	48	730	98.9%
0-0-0-0-81-e7	71,622	664	48	712	98.9%
0-0-0-0-81-ea	71,622	608	48	656	99.0%
0-0-0-0-81-f1	71,622	654	48	702	98.9%
0-0-0-0-81-f7	71,622	468	48	516	99.2%
0-0-0-0-82-14	71,622	16	48	64	99.8%
0-0-0-0-82-2f	71,622	582	48	630	99.0%
0-0-0-0-82-35	71,622	522	48	570	99.1%
0-0-0-0-82-5b	71,622	664	48	712	98.9%
0-0-0-0-82-5d	71,622	676	48	724	98.9%
0-0-0-0-82-6	71,622	692	48	740	98.9%
0-0-0-0-82-71	71,622	1,054	48	1,102	98.4%
0-0-0-0-82-77	71,622	944	48	992	98.5%
0-0-0-0-82-7d	71,622	422	48	470	99.2%
0-0-0-0-82-82	71,622	500	48	548	99.1%
0-0-0-0-82-8d	71,622	676	48	724	98.9%
0-0-0-0-82-8e	71,622	502	48	550	99.1%
0-0-0-0-82-91	71,622	622	48	670	99.0%
0-0-0-0-82-92	71,622	612	48	660	99.0%
0-0-0-0-82-95	71,622	638	48	686	98.9%
0-0-0-0-82-9b	71,622	564	48	612	99.0%
0-0-0-0-82-9d	71,622	592	48	640	99.0%
0-0-0-0-82-a3	71,622	622	48	670	99.0%
0-0-0-0-82-a4	71,622	678	48	726	98.9%
0-0-0-0-82-b	71,622	624	48	672	99.0%
0-0-0-0-82-b0	71,622	628	48	676	99.0%
0-0-0-0-82-c9	71,622	674	48	722	98.9%

Table 7.4: Results from Experiment 3, Part 2 of 3

badge watched	*unfiltered event msgs*	*actual event msgs*	*overhead msgs*	*total msgs*	*relative decrease*
0-0-0-0-82-ca	71,622	486	48	534	99.2%
0-0-0-0-82-d	71,622	678	48	726	98.9%
0-0-0-0-82-ea	71,622	656	48	704	98.9%
0-0-0-0-82-ee	71,622	586	48	634	99.0%
0-0-0-0-82-f0	71,622	12	48	60	99.8%
0-0-0-0-83-11	71,622	726	48	774	98.8%
0-0-0-0-83-1d	71,622	534	48	582	99.1%
0-0-0-0-83-25	71,622	118	48	166	99.7%
0-0-0-0-83-27	71,622	682	48	730	98.9%
0-0-0-0-83-29	71,622	706	48	754	98.8%
0-0-0-0-83-2a	71,622	698	48	746	98.9%
0-0-0-0-83-31	71,622	68	48	116	99.7%
0-0-0-0-83-36	71,622	660	48	708	98.9%
0-0-0-0-83-38	71,622	516	48	564	99.1%
0-0-0-0-83-39	71,622	686	48	734	98.9%
0-0-0-0-83-3c	71,622	546	48	594	99.1%
0-0-0-0-83-3d	71,622	550	48	598	99.1%
0-0-0-0-83-40	71,622	684	48	732	98.9%
0-0-0-0-83-42	71,622	550	48	598	99.1%
0-0-0-0-83-48	71,622	4	48	52	99.8%
0-0-0-0-83-49	71,622	662	48	710	98.9%
0-0-0-0-83-4d	71,622	558	48	606	99.1%
0-0-0-0-83-4e	71,622	104	48	152	99.7%
0-0-0-0-83-5	71,622	558	48	606	99.1%
0-0-0-0-83-52	71,622	642	48	690	98.9%
0-0-0-0-83-58	71,622	648	48	696	98.9%
0-0-0-0-83-5d	71,622	664	48	712	98.9%
0-0-0-0-83-63	71,622	4	48	52	99.8%
0-0-0-0-83-80	71,622	436	48	484	99.2%
0-0-0-0-83-95	71,622	100	48	148	99.7%
0-0-0-0-83-c5	71,622	604	48	652	99.0%
0-0-0-0-83-d3	71,622	568	48	616	99.0%
0-0-0-0-83-dd	71,622	268	48	316	99.5%
0-0-0-0-83-e8	71,622	434	48	482	99.2%
0-0-0-0-83-e9	71,622	52	48	100	99.8%
0-0-0-0-83-f	71,622	684	48	732	98.9%
0-0-0-0-83-f7	71,622	338	48	386	99.4%
0-0-0-0-84-1a	71,622	188	48	236	99.6%
0-0-0-0-84-1f	71,622	444	48	492	99.2%
0-0-0-0-9-fd	71,622	446	48	494	99.2%
0-0-0-0-a-cd	71,622	4	48	52	99.8%
0-0-0-0-d-32	71,622	8	48	56	99.8%
0-0-0-0-d-3c	71,622	76	48	124	99.7%
0-0-0-0-e-93	71,622	388	48	436	99.3%
0-0-0-0-e-ae	71,622	16	48	64	99.8%
0-0-0-0-e-b2	71,622	384	48	432	99.3%
0-0-0-0-e-b9	71,622	402	48	450	99.3%
0-0-0-0-e-cc	71,622	814	48	862	98.7%
0-0-0-0-e-cf	71,622	24	48	72	99.8%
0-0-0-0-e-de	71,622	484	48	532	99.2%
0-0-0-0-e-e9	71,622	1,096	48	1,144	98.3%
0-0-0-0-e-ec	71,622	386	48	434	99.3%
0-0-0-1-30-ee	71,622	128	48	176	99.7%
0-0-0-1-38-a8	71,622	150	48	198	99.6%
Average					99.2%

Table 7.5: Results from Experiment 3, Part 3 of 3

Badges watched	*Unfiltered event msgs*	*Actual event msgs*	*Overhead msgs*	*Total msgs*	*Relative Decrease*
Users	71,622	19,868	48	19,916	72.2%
Equipment	71,622	47,156	48	47,204	34.1%
Unlisted	71,622	4,598	48	4,646	93.5%
Average					66.6%

Table 7.6: Results from Experiment 4

7.3.4 Experiment 4

As described in section 6.1, the badges in the badge system can belong to either people or equipment. The *big brother* entity used in this experiment uses this characteristic to subscribe to sightings of one type of badge or the other. The entity works by creating a notify constraint which reads information from a list of badges stored in the shared file system. The list contains information as to which badge identifiers belong to equipment and which to personnel. In the experiment, three big brother entities were created, one for each type of badge (user or equipment) and one for unlisted badges, i.e., badges without entries in the database. The results are shown in table 7.6.

As can be seen from the table, a big brother entity on average receives a third ($100\% - 66.6\%$) of the messages which would have been sent if filters had not been in use. This is a fair decrease in bandwidth usage. This number of messages (one third) is hardly surprising, considering that the three big brother modes separate the events into three groups, the sum of which constitute all the events raised. For practical purposes, the last big brother mode may be particularly useful as a means to continuously monitor for unlisted badges and thereby check whether the badge information database is up to date. Assuming that the database is relatively up to date, this would be a useful and cheap (in terms of overhead) use of filters.

7.4 Dynamic Linking Measurements

In objective B (section 1.6.2), we outlined a number of goals with respect to dynamic linking support in the SECO implementation. They were,

1. to provide support for dynamic linking
2. to determine how much extra complexity (in terms of code) is needed
3. to measure how large the gain is in terms of decreased application footprint

In this section, we review each of these goals, present the relevant data, and discuss to what degree the goal has been fulfilled. The approach is to present and discuss measurements of code sizes from the experiments presented in section 7.3.

7.4.1 Dynamic Linking Support

Each of the four experiments described in section 7.3 featured an entity subscribing to events using a notify constraint object belonging to a dynamic class. The application instance holding the subscriber registered the dynamic class with its local class repository before performing the subscription. The other application instances dynamically linked the notify constraint code as events were raised and needed to be matched against the constraint. The SECO implementation can therefore be said to include support for dynamic linking. This fulfills the first part of Objective B, as defined in section 7.4.

Class Versioning

Class versioning can be used to change an application gradually by introducing new versions of classes at runtime. Since the constraints used in the active badge system simulation did not change, that application did not use this feature. Evaluating class versioning in particular was not part of Objective B.

7.4.2 Source Code Complexity

By requiring the application to include code for dynamic linking support, overhead is imposed on the application programmer. This overhead has the form of extra lines of source code which mean that development time (and, consequently, cost) is increased. In this section, we measure the overhead by counting the number of lines of source code used for dynamic linking and comparing it with the total amount of source code for each subscriber entity used in the four experiments from section 7.3. The data is shown in table 7.7 on page 108.

As can be seen from the table, the code required for dynamic linking on average constitute more than half of the code required in total for each particular entity. Hence, for these four entities there is typically more code for implementing dynamic linking support than there is for implementing the constraint itself. Whether this is an important overhead depends on the means by which the code is generated. If handwritten, there is obviously work involved and dynamic linking is then fairly expensive. However, the code was

Constraint	*Constraint Source Code Size*	*Lines Needed for Dynamic Linking*	*Dynamic %*
God	203 lines	122 lines	60%
SecurityCam	227 lines	122 lines	54%
PrivateEye	227 lines	122 lines	54%
BigBrother	256 lines	128 lines	50%
Average	228 lines	124 lines	54%

Table 7.7: Dynamic Linking's Influence on Application Source Code Size

originally designed to be generated by a preprocessor (DCLASS, described in section 5.1.3) and can be assumed to be automatically generatable by a tool of the right kind. Such a tool could be an *entity editor* type program which could, for example, include a graphic user interface. A tool of this kind would ease application development tremendously and, if available, could probably fairly easily be extended to generate code with support for dynamic linking.

We have determined that there is a substantial overhead involved in using dynamic linking, amounting to more than half the lines of source code in the four example programs.

7.4.3 Footprint Decrease

Another issue with regards to dynamic linking support is the influence it has on the application footprint. Replacing real code with stub code during compilation can decrease application footprint if the stub code is smaller than the real code and the real code is never linked. However, dynamic linking also comes at a price, since extra code (in form of stub code and the class repository) is needed to perform the dynamic linking. This section presents and discusses the costs and benefits of dynamic linking with regards to application footprint by looking at the sizes of the compiled object files. (Details of the compiler configuration can be found in section 7.2.1.) Support for dynamic linking falls into two parts: the class repository and the stub/upcall code. In the following, we look at each in turn.

Class Repository Overhead

In order to use dynamic classes, the application must be linked with the class repository library. The size of the object code in this library is 169,237 bytes which forms a basic cost that can only be avoided if the application does not use dynamic linking at all.[6] The first dynamic class is therefore quite

[6]In the current implementation of SECO this implies not using notify constraints, since notify constraint classes have to be dynamic.

Constraint	*n1*	*n2*	*n3*	*Change (not linked)*	*Change (linked)*
God	37,750	31,366	79,376	−16.9%	+110%
SecurityCam	41,895	31,506	83,666	−24.8%	+99.7%
PrivateEye	41,880	31,476	83,620	−24.8%	+99.7%
BigBrother	42,278	33,170	85,712	−21.5%	+103%
Average				−22.0%	+103%

Table 7.8: Influence of Dynamic Linking on Class Footprint

expensive, since it also includes the overhead of the entire class repository library. Subsequent dynamic classes are less expensive. In the following, we look at the cost of each class.

Stub and Upcall Overhead

We have seen that the class repository poses a basic overhead for any application using dynamic classes. In addition, each actual dynamic class has an influence on the application footprint. For a static class, this influence is simply the size of the object code implementing it. We call the size of this code the *class footprint*. For dynamic classes, the actual class footprint depends on whether the class code is ever linked or not. If no object belonging to the class is ever instantiated, the class code itself is never linked into the application. In this case, the class footprint is fairly small, since only the stub code required to link the class actually resides in the application. If, on the other hand, an object is instantiated and the class code is linked, the class footprint effectively increases. The extra code (in addition to the stub code) is the upcall code and the real class implementation.

Table 7.8 shows class footprints for the four constraint classes used in the simulation of the active badge system. Column *n1* shows the footprints for the real class code, i.e., either the code dynamically linked on instantiation of a dynamic class, or the entire class footprint had the class been implemented statically. Column *n2* shows the size of the stub code which is linked instead of the real code in column *n1*. If the class code is never linked, this is the entire class footprint. Column *n3* shows the entire footprint in case the dynamic class is linked. It is the sum of columns *n1* and *n2* plus the size of the upcall code (not shown in the table). The last two columns show the relative change in class footprint, for a class that was not linked and linked, respectively.

The table shows similar results for each of the four dynamic classes. There is a moderate decrease (average 22%) in footprint for each in case the class code is never linked. In case the class code is linked, the overhead is substantial: on the order of a factor two for each class. The data shown in the table

does not take the basic cost of the class repository library into account, so these figures can be expected to hold for subsequent dynamic classes.

Discussion

There are some advantages to dynamic linking but they come at a cost. In case the application uses only a few dynamic classes, the basic cost (in terms of footprint increase) of including the class repository code will almost certainly exceed anything gained from linking stub code instead of real code. In this case, dynamic linking is unlikely to be worthwhile as a means of decreasing application footprint, even if none of the class code is ever actually linked. In the simulation of the active badge system, there was only one (fairly small) dynamic class in each application and it was always linked, so in this particular case, dynamic linking increased rather than decreased application footprint. Other applications may implement a number of large classes, few of which will be used by the same application instance. Such applications may possibly benefit by a somewhat decreased footprint, but the simulation of the active badge system is not such an application.

A possibly more valuable use for dynamic classes is that they enable new code to be added during runtime. The simulation of the active badge system did not use this feature, but certain applications may want the ability to change incrementally. For such applications, the dynamic linking facility may be worth the cost of increased footprint, especially if the dynamic linking facility is extended to include other types of code than notify constraint code.

Though the experiments have shown dynamic linking to be perfectly possible, it seems that the benefit from using dynamic classes to decrease footprint is limited unless for fairly specialised applications. A more likely use may be for applications to change incrementally by introducing new versions of old classes. In the current version of SECO, notify constraint classes are required to be dynamic. Since some applications (e.g., the active badge system) will suffer from a substantial increase in footprint by using dynamic classes, it would be attractive to make dynamic linking optional. Further work on dynamic class support in SECO (e.g., to extend support to other types of code) should take this into account.

We have measured the gain of dynamic linking in terms of decreased application footprint. Though the gain itself was not impressive, this fulfills part three of Objective B, as described in section 7.4.

7.5 Conclusions

This chapter has presented the experiments performed with the SECO implementation. A distributed simulation of the Cambridge active badge system was used for various purposes. First, it was used to show that the SECO implementation is usable and that it fulfills the elementary goals of distribution and dynamic linking. Second, it was used to measure the savings in terms of reduced network overhead for a series of filters likely to be usable in a real active badge system. Third, the simulation source and binary files were examined in order to discover the price of dynamic linking.

We found that filtering was generally worthwhile in the example simulation. Notify constraints caused a reduction of between 99.9% and 34.1% for all entities used in the experiments, except the *God* entity where notify constraints caused a slight increase. To what extent these findings can be expected to hold for other applications of course depends on the applications in question. The application in this scenario used subscriptions which were in effect for a fairly long time. Applications with frequent subscriptions (and cancelling of subscriptions) will benefit less from using notify constraints, but for the active badge system constraints were extremely useful.

Dynamic linking was in general found to be very expensive, in terms of increased source code complexity and application footprint alike. The examination of footprints showed that it is doubtful (at best) whether this feature can be used to increase the scalability of applications, perhaps with the exception of very specialised ones. The vastly increased source code requirements means that dynamic classes are likely to need to be generated by an external tool, such as an entity editor, in order to be feasible.

The three objectives from section 1.6 were shown to have been fulfilled. Section 7.3 fulfilled objectives A and C and section 7.4 fulfilled objective B.

Chapter 8

Conclusion

This thesis has the described the design and implementation of a distributed version of the ECO model and evaluated two techniques for increasing scalability. In many ways, the work presented herein has relied heavily on the discussions in chapter 1, especially with regards to the intangible concept of scalability and the division of the overall goal into the three subgoals. In this chapter, we attempt to place the results in a greater context by summing up the conclusions made in chapter 7 and drawing a line back to the initial discussions on scalability and event models. We then take a retrospective look at the division of the overall goal in three before finally listing the most interesting possibilities for future work.

8.1 Scalability in the ECO Model

The discussion about scalability in section 1.5 identified four parameters in the ECO model which could in principle be scaled independently. We claimed that despite the theoretical independence of the parameters, there is likely to be a relationship between them in any real system. Two parameters (number of users and entities) could be scaled by scaling the third (number of nodes) at the cost of decreased scalability of the fourth (activity). In any large-scale distributed event system, *activity* is probably the parameter which is most difficult to scale. New nodes can be added practically ad inifinitum but they all have to exchange events over the same network and as activity grows, the network easily becomes a bottleneck. Reducing network traffic is therefore an important way of scaling activity in any such system.

This thesis has shown filtering to be an extremely powerful means to reduce network traffic in an event-based system, and consequently a feasible way to dramatically increase scalability. This is the most important finding

of this thesis. Furthermore, because the experiments were conducted with data from a real event-based system, we claim the results to have practical relevance and expect them to hold for similar event-based systems outside the laboratory. Indeed, a sign that the industry is becoming aware of the importance of event filtering is the OMG's initiative to augment their event service with filtering capabilities [Gro96].

The other scalability technique, dynamic linking, was found to be much less valuable for the event-based system, and our conclusion is that dynamic linking is hardly worthwhile as a means to scale such systems. Dynamic linking may, or may not, have other virtues for ECO (such as increased dynamism), but we have found that scalability is not one of them.

As stated in section 1.6, the overall objective of this thesis has been to evaluate two scalability techniques—filtering and dynamic linking—in the ECO model. The overall conclusion is that ECO can be made to scale very well by using notify constraints but not by using dynamic linking.

8.2 Conclusion Validity

In the introduction to this thesis, section 1.6 claimed that the overall objective was too abstract to be practical and therefore divided it into three subgoals, *Objective A, B,* and *C*. Now, at the end of the project, it is worth reviewing this division to see if it was successful and whether or not it makes as much sense from a conclusive as from an introductory viewpoint.

Chapter 7 made three conclusions corresponding to the three objectives. Briefly stated, they were,

Objective A: The ECO model can be implemented in a fully[1] distributed fashion. That we were able to run the experiments using the SECO system shows the implementation to be perfectly usable as a research environment.

Objective B: Dynamic linking is possible but as a means of increasing scalability by decreasing application footprint it is hardly worthwhile. This was shown by the fact that there was a large overhead in terms of extra code required to support dynamic linking, and the resulting decrease in footprint size was small.

Objective C: Filtering proved to be a very powerful way of increasing the

[1]The SECO implementation currently contains two centralised components which have shown to be replacable with distributed counterparts.

scalability of a system. For long-lasting subscriptions, massive reductions in network traffic were achieved at a small cost.

Section 1.6 identified *Objective A* to be a prerequisite for the two other objectives. We could not have evaluated dynamic linking and filtering in a convincing manner without access to a distributed implementation of the event model, so this has shown to hold true. We also claimed *Objectives B* and *C* to be independent, meaning that it be possible to evaluate them separately. This has also held true and can be seen from the fact that we independently reached vastly different conclusions for these two objectives. All in all, the division of the overall goal into the three subgoals was beneficial. It helped structure the task and made it more managable, and it also made it easier to present the findings in a comprehensible manner.

8.3 Future Work

This section describes the possible future work on SECO and within scalable event systems in general, based on the conclusions of this thesis.

SECO

SECO could be extended in many ways. The most vital improvement already mentioned would be to complete the work on multicast support in the communications package KANGA and to modify SECO to use it. Another improvement mentioned before would be to implement distributed versions of the two centralised components, the AIR and the shared file system.

The model's scalability could possibly be increased further by extending SECO to support *zones*, as described by [O'C97]. Also, if dynamic linking is still desired, perhaps for its ability to incrementally add code to the application rather than decrease application footprint, a future project could be to automate the generation of code required to support dynamic linking. This could either be done by repairing the DCLASS preprocessor [Fur97] or, perhaps better, by modifying the ECO entity editor [McG96] to generate the code automatically.

One of the three event models examined in chapter 2 was the Cambridge model which included a new concept called *composition*. A really interesting project would be to extend SECO to support composite filters. Different approaches to this problem could be taken, either solving it within the model, possibly extending the ECO model as a result, or by implementing composition at the application level and retaining the event model in its current form.

Event Systems and Virtual Worlds

Viewed in a greater perspective—in particular in the context of event-based virtual worlds—the success of filtering as a means to increase scalability is the most important conclusion of this thesis. In section 1.1, we stated that the problem of scalability was perhaps the greatest challenge to be faced by virtual world researchers within the immediate future. This thesis has shown event filtering to be extremely useful in dealing with this problem, and there is no doubt that this also holds for other application domains. Filtering should be a central part of future work within any scalable event system.

Though very effective, filtering on its own is unlikely to solve all scalability problems for event systems. Another interesting idea is the concept of *zones*, as described by [O'C97]. In the context of virtual worlds, zones were used to partition a world into smaller and more manageable units. For entities, belonging to one or more zones, the zones were used as 'areas of interest.' Zones were found useful by [O'C97] and future work could attempt to broaden this idea and see if it is usable for event models outside the virtual world domain.

Bibliography

[BBHM95] Jean Bacon, John Bates, Richard Hayton, and Ken Moody. Using Events to Build Distributed Applications. In *Proceedings of the 1995 Second International Workshop on Services in Distributed and Networked Environments (SDNE'95)*. University of Cambridge Computer Laboratory, 1995.

[BHJ+87] Andrew Black, Norm Hutchinson, Eric Jul, H. Levy, and L. Carter. Distribution and abstract types in emerald. In *IEEE Transactions on Software Engineering, vol. SE-13*, 1987.

[Bur96] Garrett Burke. Kanga: A framework for building application specific communication protocols. Master's thesis, Department of Computer Science, Trinity College Dublin, September 1996.

[BYG89] Richardo A. Baeza-Yates and Gaston H. Gonnett. Automaton Searching on Tries or Fast Text Searching for Regular Expressions. In *ICALP'89, Lecture Notes in Computer Science 372*, pages 46–62. Springer-Verlag, Stresa, Italy, July 1989.

[CADA87] R. L. Cooper, M. P. Atkinson, A. Dearle, and D. Abderrahmane. Constructing database systems in a persistent environment. In Peter M. Stocker and William Kent, editors, *Prooceedings of the Thirteenth International Conference on Very Large Databases*, pages 117–126, Brighton, 9 1987.

[CDK94] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems, Concepts and Design.* Addison-Wesley, Workingham, England, 2nd edition, 1994.

[Fur97] Anders Furuhed. A Class Register for the Tigger Framework. Master's thesis, Kungl Tekniska Högskolan, Sweden, 1997.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* Addison-Wesley, 1996. ISBN 0-201-63451-1.

[Gro95a] Object Management Group. *The Common Object Request Broker: Architecture and Specification, V2.1.* Object Management Group, 1995.

[Gro95b] Object Management Group. *CORBAservices: Common Object Services Specification*, chapter 4. Object Management Group, 1995.

[Gro96] Object Management Group. Notification service, request for proposal, December 1996.

[Hay96] Richard Hayton. *OASIS, An Open Architecture for Secure Internetworking Services.* PhD thesis, University of Cambridge, March 1996. Technical Report TR399.

[HBBM96] Richard Hayton, Jean Bacon, John Bates, and Ken Moody. Using Events to Build Large Scale Distributed Applications. In *Proceedings of the Seventh ACM SIGOPS European Workshop*, pages 9–16. Association for Computing Machinery, September 1996.

[McG96] Stephen McGerty. The entity editor — a tool for defining entity behaviour. Master's thesis, Department of Computer Science, Trinity College Dublin, October 1996.

[Mic97] Sun Microsystems. Javabeans api specification, version 1.01, July 1997.

[O'C97] Karl O'Connell. *System Support for Multi-User Distributed Virtual Worlds.* PhD thesis, Trinity College, Department of Computer Science, Dublin 2, Ireland, October 1997. (Submitted).

[ODC+96] Karl O'Connell, Tom Dinneen, Steven Collins, Brendan Tangney, Neville Harris, and Vinny Cahill. Techniques for Handling Scale and Distribution in Virtual Worlds. In *Proceedings of the Seventh ACM SIGOPS European Workshop*, pages 17–24. Association for Computing Machinery, September 1996.

[Orw90] George Orwell. *1984.* New American Library, 1990. ISBN 0-451-52493-4.

[SCT95] Gradimir Starovic, Vinny Cahill, and Brendan Tangney. An Event Based Object Model for Distributed Programming. In John Murphy and Brian Stone, editors, *Proceedings of the 1995*

International Conference on Object Oriented Information Systems, pages 72–86, London, December 1995. Dublin City University, Ireland, Springer-Verlag.

[SDE+98] BEA Systems, DSTC, Expersoft, Fujitsu, GMD Fokus, IBM, ICL, IONA, NEC, Nortel, Oracle, TIBCO Software, and Visigenic Software. Notification service, joint revised submission. OMG, January 1998.

[SPFA94] Marc Shapiro, David Plainfoss, Paulo Ferreira, and Laurent Amsaleg. Some key issues in the design of distributed garbage collection and references. In *Unifying Theory and Practice in Distributed Systems*, Dagstuhl (Germany), September 1994.

[Str91] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, 1991. ISBN 0-201-53992-6.

[Tay95] Paul Taylor. The roo thread library. Technical report, Distributed Systems Group, Department of Computer Science, Trinity College Dublin, 1995.

[Tea95] TCD Moonlight Team. VOID shell specification. Project Deliverable Moonlight Del-1.5.1, Distributed Systems Group, Department of Computer Science, Trinity College, Dublin 2, Ireland, March 1995. Also technical report TCD-CS-95-??, Dept. of Computer Science, Trinity College Dublin.

[yDC98] Telefónica Investigación y Desarrollo and Hewlett-Packard Company. Joint submission to notification service rfp. OMG, February 1998.

[ZC95] Chris Zimmermann and Vinny Cahill. Roo: A framework for real-time threads. In *Proceedings of the 3rd Workshop on Parallel and Distributed Real-Time Systems*, pages 137–146, Los Alamitos, April 1995. IEEE Computer Society Press. Also technical report TCD-CS-95-10, Dept. of Computer Science, Trinity College Dublin.