FAULT-ADAPTATION FOR SYSTEMS IN UNPREDICTABLE ENVIRONMENTS

BY

DANIEL CHARLES STURMAN

B.S., Cornell University, 1991

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign,1994

Urbana, Illinois

ABSTRACT

This thesis describes a technique for designing systems which can adapt to patterns of faults as they occur. Such *adaptively dependable* systems are especially critical in isolated systems which must maintain long-term fault tolerance and in open systems where future system configurations are unknown. Through the use of a reflective language structure, we support dynamic installation and composition of fault-tolerance protocols. These protocols are constructed as operations on messages to allow reuse and transparency: applications and protocol may be developed separately without prior knowledge of their future composition. We have implemented our techniques in the language *Screed* which runs on the actor platform *Broadway.* Details of the implementation are also described.

ACKNOWLEDGEMENTS

I would like to thank my advisor, Gul Agha, for supporting me in this work. I would also like to thank the other members of the Open Systems Laboratory for their input and contributions. In particular, I would like to thank Svend Frølund and Rajendra Panwar for their many comments on Broadway and Screed as well as their contributions to the reflective model. Finally, I would like to thank my parents and Lily, who provide an invaluable source of support with their encouragements and love.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

Introduction

1.1 Introduction

The *failure semantics* of a service refers to the set whose elements are the different ways in which a service can fail [9]. This set serves as a contract with clients who use the service: failure semantics for a client are often dependent on the failure semantics of the services used. Therefore, an incorrect specification of the failure semantics of one service can lead to a chain of incorrect failure semantics.

For example, a data-base object may have *omission-only* failure semantics: any reply to a query will be correct, but a reply is not guaranteed. Consider an accounting program which assumes any results received from the data-base are correct. The accounting program will balance records and trigger error-finding services if the records do not balance. Assume the accounting program's failure semantics specify that it should be conservative in invoking the error-finding services: if a possible error exists, the services should be invoked. If the data-base violates its failure semantics by replying with corrupt data that hides an error, the accounting program may incorrectly deduce a correct balance and never invoke the error-finding services, thereby violating the accounting program's failure semantics.

Failure semantics are enforced through the use of *dependability protocols* which guarantee an acceptably small probability that a failure whose type is not specified may occur. Generally, the more specific the failure semantics, the greater the computational cost of the protocols. However, less stringent failure semantics for commonly used services may lead to a greater number of protocols necessary to enforce failure semantics on other services.

Many systems have been developed to support the development of dependable computing applications. In most of these systems, however, the failure semantics of a service are assumed to be static and, therefore, the dependability protocols used are fixed at compile time. However, in many computer systems, it is either unsatisfactory to adhere to a static set of failure semantics or impossible to adequately enforce the semantics with a fixed group of dependability protocols. In some systems, the protocols chosen may later prove incapable of correctly enforcing the specified failure semantics. In others, the introduction of new services may require the restructuring of failure semantics. We illustrate these situations with two example systems:

- Consider an embedded system which is required to function over a long duration, yet is fault-prone due to the uncertain environment in which it operates. If this system is physically isolated, such as in the control system of a satellite, physical modification of system components is almost impossible. In such a system, a change in the physical environment may result in protocols designed for the old environment failing to uphold the failure semantics in the new environment. New, more rigorous, dependability protocols may then be required to enforce the desired failure semantics of the system.

- Consider an *open system*. In open systems, processes may be created and removed as necessary. Creation of processes may be conditional on other events and, therefore, determining the system configuration statically may be impossible. Pre-existing services may require a stricter set of failure semantics for a new client than was originally expected. Conversely, the existing clients may allow a new service to maintain less stringent failure semantics. If this flexibility is not exploited, significant performance may be lost. Therefore, it may be impossible to determine what failure semantics a process must have, or what protocols are necessary to enforce these semantics, until after it actually joins the system. Furthermore, the addition of new services may require a change in the failure semantics of existing components. For example, a file server may initially address safety only by check-pointing the files to stable storage. New clients added to the system, however, may require the server to also provide *persistence* and a protocol to support replication may need to be added.

In this thesis, we describe a methodology for the modular specification of systems that can adapt a system's failure semantics and set of protocols enforcing the failure semantics to match the current environment. We call such systems *adaptively dependable* systems. We present a methodology which allows the reuse and transparent installation of dependability protocols as well as their dynamic installation in a system. Our methodology, when combined with a suitably structured exception handling mechanism and fault detection, allows for the development of fault handlers which can maintain consistent failure semantics within a changing environment and can alter failure semantics as system needs change. We have provided programmer support for our methodology in the language *Screed* which is implemented on our run-time system *Broadway*.

We employ *reflection* as the enabling technology for dynamic installation of dependability protocols. Reflection means that a system can reason about and manipulate a representation of its own behavior. This representation is called the system's *meta-level*. The components of an object that may be customized at the meta-level are referred to as the *meta-architecture*. In our case, the meta-level contains a description which implements the failure semantics of an executing application; reflection thus allows dynamic changes in the execution of an application with respect to dependability.

Besides supporting dynamic installation, our meta-architecture supports transparency and reuse of dependability protocols. The meta-architecture allows protocols to be expressed as operations on abstract messages. Since the individual fields of a particular message are never examined, the same protocol may be used with different applications.

Given this technique for dynamic modification of the dependability protocols used in a system, we then describe how fault detection and exception handling may be used in conjunction with our meta-architecture to support adaptively dependable systems. We model both failures and exceptions as objects. Each type of fault which may be detected is described as a specific system exception.

We construct managers — objects with meta-level capabilities — to address system exceptions. Managers serve three purposes:

- A manager may correct for recoverable faults. The corrections allow the system to continue to function despite a fault. This role is generally referred to as performing forward error recovery.

- Managers provide failure prevention. When a manager discovers a pattern of component failures, it dynamically installs protocols which *mask* future failures or facilitate future fault-correction by expanding the set of recoverable faults. In this manner, we have taken forward error recovery one step further: rather than simply adjusting the system state, the actual dependability characteristics of the system may be modified.

- Managers support reconfiguration of the dependability protocols in a system. This may be done either to alter the system's failure semantics or to correctly enforce these semantics once the environment changes. Thus, we can develop dependable long duration systems whose fault patterns are not known at start-up time.

A prototype implementation which tests these ideas is described. Our run-time system Broadway supports our meta-architecture as well as failure detection and a set of system exceptions. On top of Broadway, we have implemented the language Screed. Screed is a prototype concurrent actor language we use to illustrate our ideas. Screed provides complementary constructs for both fault detection through exception handling and for dynamic installation of protocols through a meta-architecture. This language is presented as a demonstration of how such constructs may be added to existing languages.

This thesis is organized as follows. This chapter discusses related work and background information: in Section 1.2, we discuss other work in reflection, exception handling, and in developing languages for fault-tolerance; Section 1.3 provides background information on reflection, the Actor model, and object-orientation. Chapter 2 discusses our meta-level architecture and how it may be used to construct dependability protocols. We also discuss the effect of our meta-level architecture on protocol performance. Chapter 3 describes exception handling in Screed and how exception handling may be used in conjunction with our meta-level architecture to implement adaptively dependable systems. We then illustrate this technique with an example of a system adapting to a change in environment.

Implementation details are then discussed in Chapter 4 where we discuss pertinent details of implementing our techniques for Broadway and Screed. The appendices provide a more detailed description Broadway and Screed. Appendix A provides the *Broadway User's Manual* and Appendix B is the *Screed Programmer's Guide.*

1.2 Related Work

A number of languages and systems offer support for constructing fault-tolerant systems. In Argus [20], Avalon [13] and Arjuna [28], the concept of nested transactions is used to structure distributed systems. Consistency and resilience is ensured by atomic actions whose effect are check-pointed at commit time. The focus in [24], [7] and [5] is to provide a set of protocols that represent common communication patterns found in fault-tolerant systems. None of the above systems support the factorization of fault-tolerance characteristics from the application specific code. In [33] and [25], replication can be described separate from the service being replicated. Our approach is more flexible since fault-tolerance schemes are not only described separately but they can also be attached and detached dynamically. Another unique aspect of our approach is that different fault-tolerance schemes may be composed in a modular fashion. For example, check-pointing may be composed with replication without having either protocol know about the other.

Non-reflective systems which support customization do so only in a system-wide manner. For example, customization in a micro-kernel based system [1] affects all the objects collectively. In an object-oriented system such as *Choices* [6], frameworks may be customized for a particular application. However, once customized, the characteristics may not change dynamically. Reflection in an object-based system allows customization of the underlying system independently for each object. Because different protocols are generally required for very specific subsets of the objects in a system, this flexibility is required for implementing dependability protocols.

Reflection has been used to address a number of issues in concurrent systems. For example, the scheduling problem of the Time Warp algorithm for parallel discrete event simulation is modeled by means of reflection in [35]. A reflective implementation of object migration is reported in [32]. Reflection has been used in the Muse Operating System [34] for dynamically modifying the system behavior. However, the literature is unclear as to the effect on efficiency of this generally reflective system.

Reflective frameworks for the Actor languages MERING IV and Rosette have been proposed in [14] and [31], respectively. In MERING IV, programs may access *meta-instances* to modify an object or *meta-classes* to change a class definition. In Rosette, the meta-level is described in terms of three components: a *container*, which represents the acquaintances and script; a *processor*, which acts as the scheduler for the actor; and a *mailbox*, which handles message reception

The concept of unifying exception handling and fault detection was originally proposed in [27] and then refined in [26]. In these papers, detected failures are considered asynchronous events much as exceptional conditions are treated in distributed programming languages. Therefore, exception handling constructs provide a natural way to incorporate failure-response code into an application.

Goodenough introduced the idea of exceptions and exception handling in [17]. Since then, many different exception handling mechanisms have been proposed. Exception handling constructs have been developed for object-based languages such as Clu [21] and Ada [10]. Dony [11] describes an approach for object-oriented languages which was implemented in Smalltalk. This was the first approach that implemented exceptions as objects. Exception handling for C++ is discussed in [30]. A good overview of techniques proposed for other object-oriented languages can be found in [12].

A critical difference between object-oriented approaches to exception handling and non-object-oriented approaches such as CLU [21] or Ada [10] is that, in the non-object-oriented approaches, the exception object is represented by a set of parameters to a function. Therefore, on generating the signal, this parameter list must provide all possible information used by the handler.

For concurrent systems, an approach has been proposed for languages which use RPC communication [8]. However, the technique is based on synchronized components which allows their constructions to be closer to that of a sequential system than an asynchronous system.

Exception handling mechanisms have been proposed for other Actor languages. In [19], an exception handling mechanism was proposed for ABCL/1 and for Acore in [23]. These languages use *complaint addresses* to support exception handling. A complaint address is an address, specified with each message, to which all signals are dispatched. Complaint address-based schemes do not allow for different handlers to be bound to different exceptions: all complaints are sent to the same destination. Such an approach makes supporting adaptive dependability, where each handler is an "expert" on a particular group of exceptions, very difficult to implement. Furthermore, these complaint address schemes pass exception information in terms of a set number of parameters in a message rather than utilizing exception objects.

1.3 Background

Before discussing our meta-architecture and how we use it to support adaptively dependable systems, we first discuss in greater detail some of the concepts used in this paper. Specifically, to illustrate our concepts we have chosen the Actor model of concurrent computation. The Actor model was chosen due to the ease in which our reflective architecture could be incorporated. We then briefly discuss some of the advantages of object-oriented programming and how they are important to our methods. Finally, we give a more in-depth discussion of reflection and how it relates to a programming language.

1.3.1 The Actor Model

We illustrate our approach using the *Actor model* [2, 3]. It is important to note that the idea of using reflection to describe dependability is not tied to any specific language framework. Our methodology assumes only that resources can be created dynamically, if needed, to implement a particular protocol and that the communication topology of a system is reconfigurable. Our methodology does not depend on any specific communication model.

Actors can be thought of as an abstract representation for the computing components in a multicomputer architecture. An actor is an encapsulated entity that has a local state. The state of an actor can only be manipulated through a set of *operations*.

Actors communicate by asynchronous point-to-point message passing. A message is a request for invocation of an operation in the target actor. Messages sent to an actor are buffered in a *mail queue* until the actor is ready to process the message. Each actor has a system-wide unique identifier which is called a *mail address*. This mail address allows an actor to be referenced in a location transparent way. In order to abstract over processor speeds and allow adaptive routing, preservation of message order is not guaranteed.

The *behavior* of an actor is the actions performed in response to a message. These actions may include the dynamic creation of new actors. Actor addresses may be communicated in messages and an actor may only communicate with another actor if it has the correct address. An actor's *acquaintances* are the mail addresses of known actors. An actor can only send messages to its acquaintances, which provides locality.

1.3.2 Object Orientation

In an object-oriented environment, a program execution is organized as a collection of objects. Each object is an encapsulated entity, similar to an instance of an abstract data type. The local data comprising each object may only be accessed through an interface specified as a set of *methods*. The operations carried out by a method are not visible outside the object. Objects communicate with messages which invoke a method in the receiving object. The local data of another object can not otherwise be changed or accessed.

Objects are *instantiated* from *classes*. A class is a user-defined abstraction. Classes may be thought of as types and objects as elements of that type. Instantiation is the creation of an object of a particular class. Classes contain the description (code) of the methods and of the instance variables for objects instantiated from that class. Classes may *inherit* from other classes. Inheritance provides the inheriting class with the properties – the methods and instances – of the *ancestor* class. The inheriting class can then utilize these properties as well as augment them with new instances variables or methods. Methods may be inherited directly or redefined, facilitating code reuse.

Object-oriented languages allows for a modular development of systems. The implementation of each component is hidden from other components: only the interface is known. In this way, a component's implementation may change without affecting other components. Code may also be reused efficiently since components share code by inheriting from a common ancestor class.

1.3.3 Reflection

Reflection means that a system can manipulate a causally connected description of itself [29, 22]. Causal connection implies that changes to the description have an immediate effect on the described object. In a reflective system, a change in these descriptions or *meta-objects* results in a change in how objects are implemented. The object for which a meta-object represents certain aspects of the implementation is called the *base object*.

Meta-objects may be thought of as objects which logically belong in the underlying run-time system. For examples, a meta-object might control the message lookup scheme that would map incoming messages to operations in the base object. Another meta-object may modify how values are read from memory. Using reflection, such implementation-level objects can be accessed and examined, and user defined meta-objects may be installed, yielding a potentially customizable run-time system within a single language framework.

The reflective capabilities which are provided by a language are referred to as the *meta-level architecture* of the language. The meta-level architecture may provide variable levels of sophistication, depending on the desirable level of customization. The most general meta-level architecture is comprised of complete interpreters, thus allowing customization of all aspects of the implementation of objects. In practice, this generality is not always needed and, furthermore, defining a more restrictive meta-level architecture may allow reflection to be realized in a compiled language. The choice of a meta-level architecture is part of the language design. Customizability of a language implementation must be anticipated when designing the run-time structure. Although a restrictive meta-level architecture limits flexibility, it provides greater safety and structure. If all aspects of the implementation were mutable, an entirely new semantics for the language could be defined at run-time; in this case, reasoning about the behavior of a program would be impossible.

We limit our meta-level to only contain the aspects that are relevant to dependability. Application specific functionality is described in the form of base objects and dependability protocols are described in terms of meta-objects. Thus, dependability is modeled as a special way of implementing the application in question. Our methodology supports modularity since functionality and dependability are described in separate objects. Since meta-objects can be defined and installed dynamically, the objects in a system can dynamically change the protocols enforcing their failure semantics as system needs change. Furthermore, new dependability protocols may be defined while a system is running and put into effect without stopping and recompiling the system. For example, if a communication line within a system shows potential for unacceptable error rates, more dependable communication protocols may be installed without stopping and recompiling the entire system.

Since meta-objects are themselves objects, they can also have meta-objects associated with them, giving customizable implementation of meta-objects. In this way, meta-objects realizing a given dependability protocol may again be subject to another dependability protocol. This scenario implies a hierarchy of meta-objects where each meta-object enforces a subset of the failure semantics for the application in question. Each meta-object may be defined separately and composed with other meta-objects in a layered structure supporting reuse and incremental construction of dependability protocols.

Because installation of a malfunctioning meta-level may compromise the dependability of a system, precautions must be taken to protect against erroneous or malicious meta-objects. To provide the needed protection of the meta-level, we introduce the concept of privileged objects called a *managers*. Only managers may install meta-objects. Using operating system terminology, a manager should be thought of as a privileged process which can dynamically load new modules (meta-objects) into the kernel (meta-level). It should be observed that, because of the close resemblance to the operating system world, many of the operating system protection strategies can be reused in our design. We will not discuss particular mechanisms for enforcing the protection provided by the managers in further detail here. Because only managers may install meta-objects, special requirements can be enforced by the managers on the structure of objects which may be installed as meta-objects. For example, managers may only allow installation of meta-objects instantiated from special verified and trusted libraries. Greater or fewer restrictions may be imposed on the meta-level depending on the dependability and security requirements that a given application must meet.

Chapter 2

Meta-level Architecture for Ultra-dependability

In this chapter we introduce MAUD (**M**eta-level **A**rchitecture for **U**ltra **D**ependability) [4]. MAUD supports the development of reusable dependability protocols. These protocols may then be installed during the execution of an application. MAUD has been implemented on Broadway, our run-time environment for actors.

We begin with a discussion of MAUD's structure. We then discuss how transparency and reusability of protocols are supported by this structure followed by an example of such a protocol. We finish this section by demonstrating how MAUD also allows the composition of protocols and give an example of composition.

2.1 A Meta-Level Architecture

As previously mentioned, MAUD is designed to support the structures that are necessary to implement dependability. In MAUD, there are three meta-objects for each actor: *dispatcher*, *mail queue* and *acquaintances*. In the next three paragraphs we describe the structure of meta-objects in MAUD. Note that MAUD is a particular system developed for use with actors. It would be possible, however, to develop similar systems for most other models.

The *dispatcher* and *mail queue* meta-objects customize the communication primitives of objects so that their interaction can be adjusted for a variety of dependability characteristics. The dispatcher meta-object is a representation of the implementation of the message-send action. Whenever the base object sends a message, the run-time system calls the `transmit` method on the installed dispatcher. The dispatcher performs whatever actions are needed to send the given message. Installing dispatchers to modify the send behavior makes it possible to implement customized message delivery patterns.

A *mail queue* meta-object represents the mail queue holding the incoming messages sent to an actor. A mail queue is an object with `get` and `put` operations. After installation of a mail queue meta-object, its `get` operation is called by the run-time system whenever the base object is ready to process a message. The `put` operation on a mail queue is called by the run-time system whenever a message for the base object arrives. By installing a mail queue at the meta-level, it is possible to customize the way messages flow into the base object.

The *acquaintances* meta-object is a list representing the acquaintances of a base object. In an actor system, all entities are actors. Although they may be implemented as local state, even primitive data objects, such as integers or strings, are considered acquaintances in an actor system. Therefore, in an actor language the *acquaintances* and the *mail queue* comprise the complete state of an actor. The *acquaintances* meta-object allows for check-pointing of actors.

Meta-objects are installed and examined by means of *meta-operations.* Meta-operations are defined in the class called `Object` which is the root of the inheritance hierarchy. All classes in the system inherit from `Object`, implying that meta-operations can be called on each actor in the system. The meta-operations `change_mailQueue` and `change_dispatcher` install mail queues and dispatchers for the object on which they are called. Similarly, the meta-operations `get_mailQueue`, `get_dispatcher` and `get_acquaintances` return the meta-objects of a given actor. If no meta-objects have been previously installed, an object representing the built-in, default, implementation is returned. Such default meta-objects are created in a lazy fashion when a meta-operation is actually called.

2.2 Transparency and Reuse

By describing our dependability protocols in terms of meta-level dispatchers and mail queues, we are able to construct protocols in terms of operations on messages where we treat each message as an integral entity. There are several advantages to developing dependability protocols in this manner.

The first advantage is the natural way in which protocols may now be expressed. When dependability protocols are described in the literature, they are described in terms of operations on abstract messages, i.e. the actual contents of the messages are rarely considered. Therefore, it is logical to code protocols in a manner more closely resembling their natural language description.

Secondly, because the protocols are expressed in terms of abstract messages and because every object may have a meta-level mail queue and dispatcher, a library of protocols may be developed which may be used with any object in the system. Such a library would consist of protocols expressed in terms of a mail queue and dispatcher pair. The meta-objects may then be installed on *any* object in the system. Since the protocols deal only with entire messages, the actual data of such messages is irrelevant to the operation of the protocol. Only fields common to every message, such as source, destination, time sent, etc. need be inspected.

The libraries may also be used with other applications, allowing the reuse of dependability protocols. One set of developers could be responsible for the dependability of multiple software systems and develop a protocol library for use with all of them. Since protocols implemented with MAUD are transparent to the application, other development teams, who are responsible for development of the application programs, need not be concerned with dependability. In the final system, protocols from the library may be installed on objects in the application, providing dependability in the composed system.

2.3 A Replicated Server

In this section, we provide an example of how a protocol may be described using MAUD. In a distributed system, an important service may be replicated to maintain availability despite

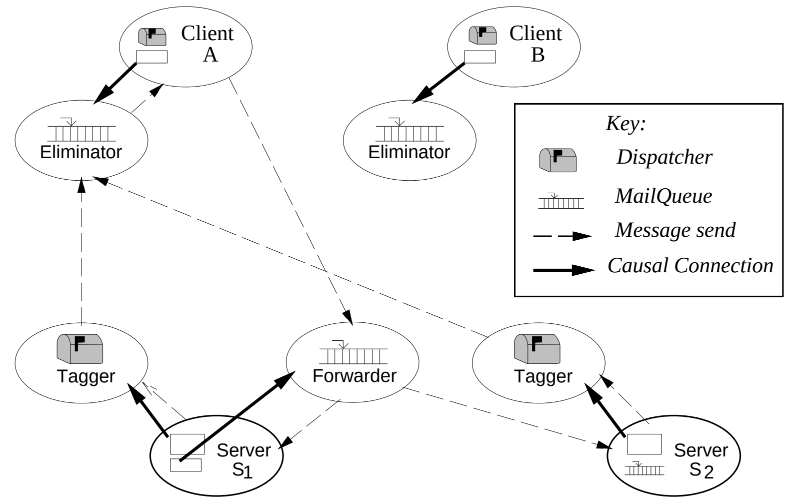

Figure 2.1: A replication protocol installed on an arbitrary server.

processor faults. In this section, we will give an example of how MAUD can be used in an actor domain to develop a modular and application-independent implementation of a protocol which uses *replication* to protect against crash failures.

```
class Forwarder : MailQueue {
  var actor backup, server;
  put(msg m) {
    m.base_send();
    m.set_dest(backup);
    m.send();
  }
}
```

Figure 2.2: Code for the server-end mail queue which implements replication.

The protocol we describe is quite simple: each message sent to the server is forwarded to a backup copy of the server. In this manner, there is an alternate copy of the server in case of a crash. Reply messages from both the original and backup servers are then tagged and the client eliminates duplicate messages.

Figure 2.1 shows the resulting actions occurring when a message is sent to the replicated service. The original server is actor S_1. When a message is received by the *Forwarder*, the message is forwarded to the backup S_2. S_2 is initialized with the behavior and state of S_1. Since they will receive the same messages in the same order, their state will remain consistent. Therefore, any replies will be identical and in the same order. The replies are tagged by the dispatchers of class *Tagger* and only the first copy of each message is passed on to the client by *Eliminator*.

Forwarding messages to the backup server is implemented using a meta-level mail queue. The Screed code for this mail queue is presented in Figure 2.2. Using a dispatcher, each reply

```
class Eliminator : Mailq {
   var
      int tag;
      actor members[2];
      actor client;
   /* No get method is required since we use
    * the default behavior inherited from Mailq */
   put(msg m) {
      int i;
      for (i := 0 to 1)
         if (m.get_src() = members[i])
            /* Since the message was from a replica,
             * we know that the first argument is a tag and
             * the second is the original message.
             */
            if (m.arg[0] < tag)
               /* Discard message */
               return;
            else if (m[0] = tag) {
               self.enqueue(m[1]);
               tag := tag + 1;
            }
   }
```

Figure 2.3: Code for the client-end mail queue which implements replication.

message of the server is tagged to allow the elimination of duplicate replies by the client. A mail queue at the client performs this duplicate elimination. The code for this mail queue is shown in Figure 2.3.

We assume that managers themselves install appropriate meta-objects realizing a given dependability protocol. Therefore, we specify the relevant dependability protocols by describing the behavior of the initiating manager as well as the installed mail queues and dispatchers. A manager in charge of replicating a service takes the following actions to achieve the state shown in Figure 2.1:

1. The specified server is replicated by a manager by creating an actor with the same behavior and state.

2. A mail queue is installed for the original server to make it act as the *Forwarder* described above.

3. The mail queues of the original clients are modified to act as the *Eliminator* described above.

4. The dispatchers of the servers are changed to tag all messages so that the *Eliminator* may remove copies of the same message.

```
add_mailq (actor aMailq) {
   if (mailq = nil) {
      self.change_mailq(aMailq);
   else mailq.add_mailq(aMailq);
}
add_dispatcher (actor aDispatcher) {
   if (dispatcher = nil) {
      self.change_dispatcher(aDispatcher);
   else dispatcher.add_dispatcher(aDispatcher);
}
```

Figure 2.4: The additional methods which allow for protocol composition.

5. Upon detection of a crash of S_1, the manager takes appropriate action to ensure all further requests to the server are directed to S_2. The manager may also create another backup at this time.

Although this example is simple, it does illustrate some of the benefits of our approach. The manager initiating the replication protocol needs no advance knowledge of the service to be replicated nor does the replicated service need to know that it is being replicated. Because the clients using the replicated service are not modified in any way, this gives us the flexibility to dynamically replicate and unreplicate services while the system is running.

2.4 Composition of Dependability Characteristics

In some cases, dependability can only be guaranteed by using several different protocols. For example, a system employing replication to avoid possible processor faults may also need to guarantee consensus on multi-party transactions through the use of three-phase commit or some similar mechanism. Unfortunately, writing one protocol which has the functionality of multiple protocols can lead to very complex code. In addition, the number of possible permutations of protocols grows exponentially, making it necessary to predict all possibly needed combinations in a system. Therefore, it is desirable to be able to compose two protocols written independently. In some cases this may not be possible due to a conflict in the semantics of the two protocols. In other cases, performance may depend greatly on the way in which two protocols are composed. For most common protocols such as replication, checksum error detection, message encryption, or check-pointing, composition is possible.

Because the meta-components of an object are themselves objects in a reflective system, there is a general solution for composing two protocols using MAUD. A simple change to the meta-operations inherited from the `Object` class, along with a few restrictions on the construction of mail queues and dispatchers, allows us to layer protocols in a general way. Figure 2.4 shows how the *add_mailq* and *add_dispatcher* methods could be expressed in terms of the other meta-operations to allow layering.

Because the current mail queue, `mailq`, and the current dispatcher, `dispatcher`, are objects, we can install meta-objects to customize their mail queue or dispatcher. By adding protocols in the above manner, the new mail queue functionality will be performed on incoming

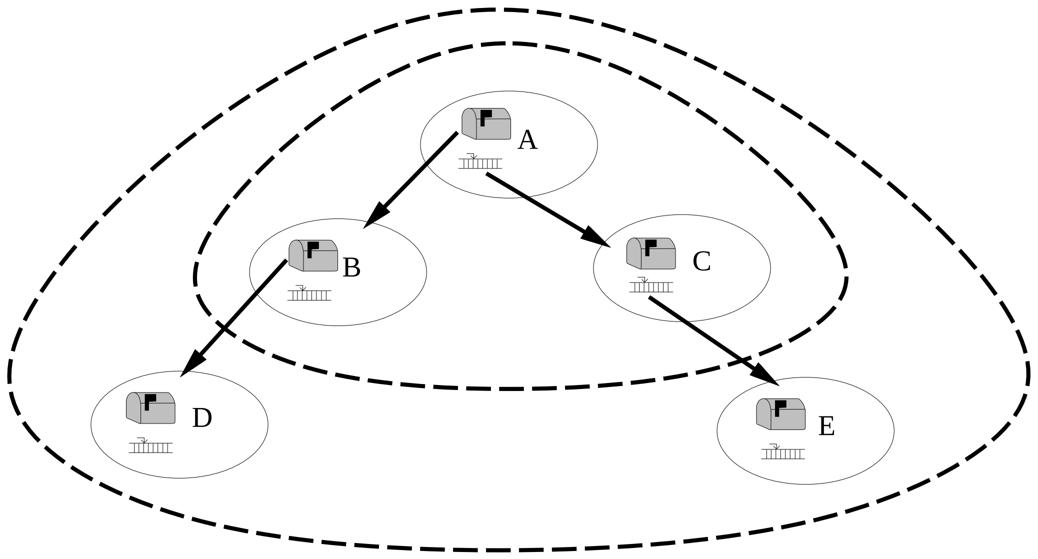

Figure 2.5: Partners and Owner relationships.

messages before they are passed on to the "old" mail queues. For the send behaviors, the process is reversed with the oldest send behavior being performed first and the newest behavior last, thereby creating an onion-like model with the newest layer closest to the outside world.

To preserve the model, however, several restrictions must be applied to the behavior of dispatchers and mail queues. We define the *partner* of a mail queue as being the dispatcher which handles the output of a protocol and the partner of a dispatcher as being the mail queue which receives input for the protocol. In Figure 2.5, B and C are partners as well as E and D. Each pair implements *one* protocol. It is possible for a meta-object to have a null partner.

The *owner application* of a meta-object is inductively defined as either its base object, if its base object is not a meta-object, or the owner application of its base object. For example, in figure 2.5, A is the owner application of meta-objects B, C, D, and E. With the above definition we can restrict the communication behavior of the actors so that:

- A mail queue or dispatcher may send or receive messages from its partner or an object created by itself or its partner.
- Dispatchers may send messages to the outside world, i.e. to an object which is not a mail queue or dispatcher of the owner application (although the message might be sent through the dispatcher's dispatcher). Dispatchers may receive `transmit` messages from their base object.
- Mail queues may receive messages from the outside world (through its own mail queue) and send messages when responding to `get` messages from their base object.
- Objects created by a mail queue or dispatcher may communicate with each other, their creator, or their creator's partner.

Because of the above restrictions, regardless of the number of protocols added to an object there is exactly one path which incoming messages follow — starting with the newest mail queue — and exactly one path for outgoing messages in each object — ending with the newest dispatcher. Therefore, when a new dispatcher is added to an object, all outgoing messages from the object must pass through the new dispatcher. When a new mail queue is installed, it will handle all incoming messages before passing them down to the next layer. Thus, a model of objects resembling the layers of an onion is created; each addition of a protocol adds a new layer in the same way regardless of how many layers currently exist. With the above rules, protocols can be composed without any previous knowledge that the composition was going to occur and protocols can now be added and removed as needed without regard not just to the actor itself, but also without regard to existing protocols. In Figure 2.5, actors B and C are initially installed as one "layer". Messages come into the layer only through C and leave through B. Therefore, D and E may be installed with the `add-mailq` and `add-dispatcher` messages as if they were being added to a single actor. Now messages coming into the composite object through E are then received by C. Messages sent are first processed by B and then by D.

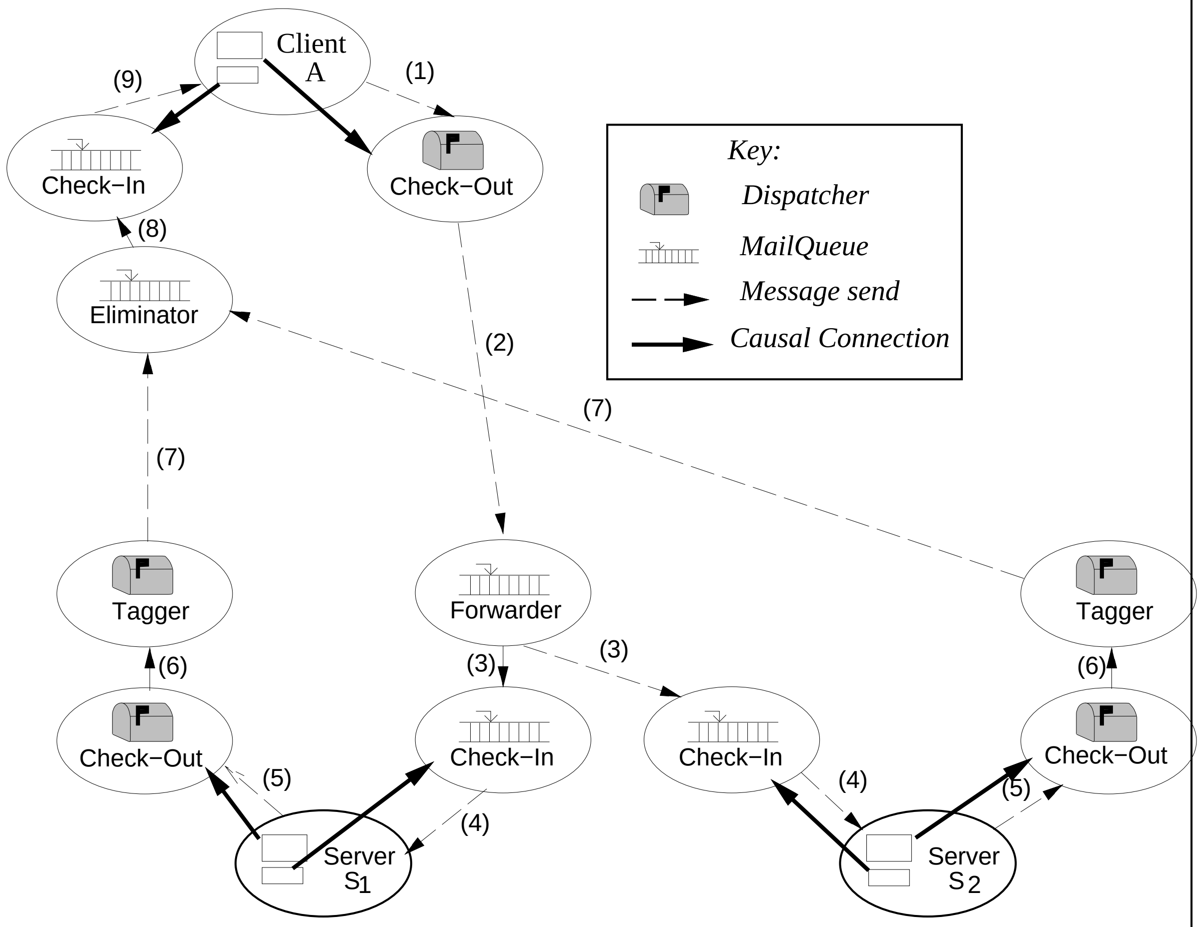

Figure 2.6: Composed system using a replication protocol and message checksum protocol.

Figure 2.6 shows the result of imposing the protocol described in Section 2.3 on a set of actors already using a checksum routine to guarantee message correctness. When a message is sent by the client A (1), the *Check-Out* dispatcher adds the checksum information to the message. The message is then forwarded to the replica as describe in Section 2.3 (2–3). The checksum information is removed by the *Check-In* mail queue(4) and the messages are processed, resulting in a reply (5). The reply messages both have the checksum (6) information added before they are tagged and sent to the client (7). At the client, duplicate messages are removed, the checksum information is checked, and the message is delivered. Although this protocol would be difficult to write as one entity, composition allows their modular, and therefore simpler, development.

Chapter 3

Exception Handling

Given a meta-level such as MAUD, it is still necessary for a programming language to provide flexible constructs for building adaptively dependable systems. In particular, it is important to convey information to the correct entities when system failures occur. We have chosen exception handling as the medium through which *managers* are informed of problems in the system. This technique has been used extensively with forward error recovery: we simply extend the notion by having our managers prevent future failures through dynamic protocol installation.

In this chapter, we describe the exception handling mechanism in Screed, our prototype actor language. Details on the general syntax of Screed may be found in Appendix B. To support adaptively dependable systems, faults and exceptions have been unified as one concept and exception handlers may be shared between objects. Broadway provides a set of system exceptions, some of which are notifications of failures. For example, when an actor which attempts to communicate with an unreachable node, a `crash` exception is generated.

We begin with a discussion of the general structure of exception handling in Screed followed by a specific illustration of the syntax used. We then show how this structure may be used with the meta-architecture to design adaptively dependable systems.

3.1 Exception Handling Components

Exceptions are *signaled* whenever an unexpected condition is encountered. An exception may be signaled either by the run-time system or by the application. The former are referred to as *system exceptions* and the latter as *user-defined exceptions*.

Exceptions in Screed are represented as objects, as proposed in [11] for sequential object-oriented languages. None of the other concurrent languages discussed in Section 1.2 have taken this approach. However, we feel this approach allows for more flexible and efficient exception handling: all the information needed by a handler is contained in one object.

All system exceptions are derived, through inheritance, from the class `exception`. User-defined exceptions may inherit from the exception class or from any other node on the system exception inheritance tree. Below, we discuss the parties involved in the generation of an exception and then the structure of system exceptions.

There are four roles involved in handling any exceptional condition: invoker, signaler, exception, and handler (see Figure 3.1). Each role is represented as an object in the system. The *invoker* initiates the method of the *signaler* which results in an exception. The occurrence of an exception generates a signal. When a signal occurs, a new *exception* object is created. The signaler notifies the appropriate *handler* object of the exception's mail address. The handler must then diagnose the exception and perform any appropriate actions for handling the exception.

In Screed, each message sent by the invoker is contained within a scope binding each possible exception to a handler. When the signaler signals an exception, an exception object of the appropriate type is created. The address of this exception object is then communicated to the appropriate handler as specified by the exception handling scope at the invoker. This communication is asynchronous, invoking a method in the handler whose name is the same as the exception's name and takes one parameter: the address of the exception object.

Exception handlers are constructed by the programmer as Screed actor-classes. For each exception a handler accepts, a method must exist with the same name as the exception and which takes an instance of the exception class as a parameter. In all other ways, handlers are identical to other actor classes: they may have a set of instance variables, inherit from other classes, and may communicate with any of their acquaintances. They may also have other, non-exception methods.

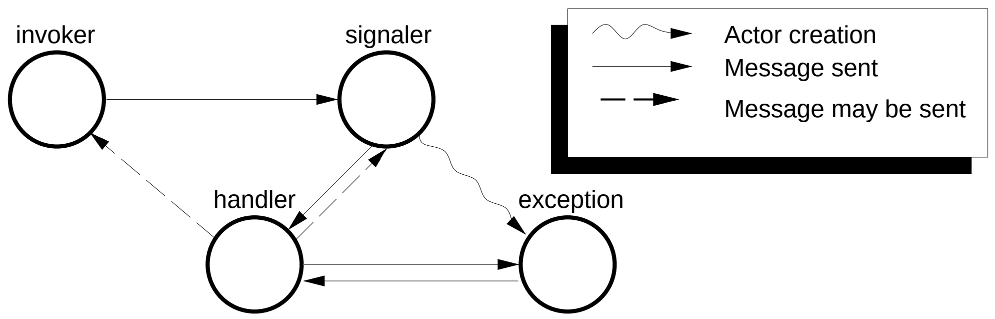

Figure 3.1: The four roles involved with an exceptional condition.

As mentioned above, when an exception is signaled, an object of the appropriate exception class is instantiated and initialized with any information needed by the handler to process the exception. Some of the initialization fields are supplied by the run-time system. These fields are contained in the `exception` class from which all exception objects inherit, and are utilized through the methods inherited from the `exception` class.

Additional arguments for the initialization of an exception may be specified by the objects raising a signal. For example, an `arithmetic` exception which is initiated by an application could be initialized when signaled with the values of the operands in the arithmetic operation. This exception object would still have default values specified by the system.

Methods defined in the exception class make use of the system-supplied values. These methods are:

`name` returns the name of the exception as a method value. Since method names are first-class values in Screed, this method enables the invocation of the correct method in the handler.

`invoker` returns the mail address of the actor which invoked the method resulting in the generation of the signal.

`signaler` returns the mail address of the signal generator.

`source` returns the name of the method in which the signal was generated.

`arguments` returns a list of the arguments that were passed to the method in which the signal was generated.

`request` returns `TRUE` if the invoker is waiting on a reply (i.e. an explicit continuation was specified), `FALSE` otherwise.

`reply` This method allows a handler to reply to a request that was interrupted by the signal. This method can be used to supply an acceptable value to the invoker, thereby allowing the continuation of the computation.

Each exception handler may utilize only a few of these fields. However, since our environment is asynchronous, we want to preserve all available information. There are no guarantees that this information will be retained by either the invoker or the signaler. Use of exception objects provides us with the flexibility to include a large amount of information without creating complicated function calls or messages: all the information is packed into an object and is referenced through a standard interface. In a procedural approach, long parameters lists would be necessary to achieve the same effect.

Broadway currently supports three different system exceptions. All three inherit directly from the class `exception`. A `bad-method` exception is instantiated when an actor receives a message it cannot processes. The `bad-method` exception class provides the behavior of the destination actor. In general, there is very little that may be done by the run-time objects to correct such an error, but this information allows a handler to provide meaningful error messages to the user.

An `arithmetic` exception is generated whenever Broadway traps an arithmetic error. Currently, this exception provides the state under which the exception occurred. We hope to soon expand this exception to include a string representing the expression being evaluated.

Broadway also provides some failure detection capabilities. Each node on Broadway has a failure detector which uses a *watch-dog timer* approach to detect the failure of, or inability to communicate with, other nodes. A `crash` exception is generated whenever an actor attempts to communicate with an actor on a non-existent or unreachable node. A `crash` exception consists of the original message and the identity of the node which cannot be reached. Notice that, although Broadway has detected a component failure, it is treated identically to any other system exception. It is also possible for an object to *subscribe* to a failure detector. In this case, the subscriber will automatically receive an exception whenever a failure is detected, even if the object never attempted try to communicate with the failed node.

Besides detecting node crashes, Broadway will also handle the failure of individual actors. If an actor crashes due to an error that is trapped by Broadway, that actor address will be marked as a crash. Currently, only arithmetic errors are trapped by Broadway and, therefore, this is the only manner in which a single actor may crash. If the defunct actor receives a message, a `dead-actor` exception will be generated. The `dead-actor` exception inherits from the `crash` exception. It also contains a reference to the exception generated when the actor crashed. (Currently, this is always an `arithmetic` exception.)

3.2 Syntax for Exception Handling

In this section, we describe our two syntax additions to Screed which enable exception handling: the `handle` statement which associates exceptions with handlers, and the `signal` statement which generates an exception.

3.2.1 Description of Handlers

```
handle (exception1,exception2 with  handler2,
        exception3 with handler2,
        ⋮
       )
{
   /* Any block of code goes here */
}
```

Figure 3.2: The structure of a handle block in Screed.

In Screed, handlers can be associated exceptions for either entire actor classes or with arbitrary code segments within a method. Figure 3.2 gives the syntax for a `handle` statement. The statement defines a scope over which specific exceptions are associated with a particular handler. If any method invocation contained within the code block of the `handle` statement results in an exception, the signal is routed to the correct handler as specified by the `with` bindings. Exceptions are specified as the name of the exception class and the handlers are addresses of objects.

Handler statements may be nested. In this case, when an exception is generated, the innermost scope is searched first for an appropriate handler. If a handler for the exception does not exist then higher level scopes are checked.

```
handle (arithmetic with arithhandler,
        bad-method with aborthandler) {
   var actor A;
   A := new complex(2,3);
   A.divide(C)(actor B) {
      handle (arithmetic with myhandler)
         B.divide(D)(actor E) {
         myNum := res;
      }
   }
}
```

Figure 3.3: An example of handler scopes and their effect.

Figure 3.3 demonstrates the scoping rules. In the scope of the outer handle statement, if in computing B (by dividing A by C), an arithmetic exception is generated (possibly by dividing by zero), the signal will be passed to `arithhandler`. The computation of E through the division of B by D, however, is in the scope of the second *handle* statement. Therefore, any arithmetic signals generated by this action are sent to `myhandler`. Conversely, if our complex objects do not have a divide method, our actions will generate a `bad-method` signal which will be handled by `aborthandler`.

Unlike the complaint address based schemes[19][23], our syntactic mechanisms do not require explicit specification of a handler's address with each message. For any given scope, including a single message send, handlers — our equivalent of complaint addresses — may be specified for each individual exception or for any group of exceptions. One handler need not be specified for all exceptions. Additionally, our method takes greater advantage of the available inheritance mechanisms as well as the general structure of object-oriented languages: both exceptions and handlers are expressed as objects in our system.

The above constructs work well within methods. However, there are two levels of scoping above the method level in Screed: the class and global levels. Exception handling at the class level is specified through the use of a `handler` statement which encloses several method definitions. In this manner, exception handling may be specified for an entire class by enclosing all methods in one handler statement. Such a construction does not prohibit `handler` statements inside the methods.

A `handle` statement may *not* be defined across class boundaries as that would require the use of shared variables between class instances. However, at the global level, Screed has a system-defined handler class called `Default-Handler`. An instance of this class handles all signals which are not caught by another handler. Default system behavior is for a signal to be simply reported to the terminal. `Default-Handler` may be overwritten by a programmer defining a custom class of the same name. In this way, a final level of exception handling may be defined by the programmer. This type of facility is especially useful for writing debuggers. Any exception not defined in a custom `Default-Handler` class is handled by the system-default. Note that the system creates only one instance of the `Default-Handler` class: all otherwise unhandled signals are delivered to this instance.

3.2.2 Signal Generation

As mentioned previously, exceptions may be generated as user-defined signals. A signal is generated by a `signal` statement.

`signal` *exception-class-name(args ...)*;

The `signal` statement sends a message to the appropriate exception handler. The arguments are used for initialization of the exception as defined by the interface of the particular exception class. The `signal` is an asynchronous message send and does not interrupt the flow of control in the code.

In many cases, it is necessary for the signaler of the exception to await a response from the handler before proceeding. `signal` statements are treated, syntactically, as message sends to a handler. Therefore, explicit continuations may be specified for `signal` statements as they are specified for Screed message-sends. In this manner, the handler may return a value to be used by the signaler. Such a case would be:

```
signal div-zero()(actor res) {
    ... some use of res...
```

```
        }
```
For this example, the exception handler would return an actor address as the value *res*. The rest of the signalling method will compute. However, the explicit continuation will be executed upon return of the result.

In other systems, a special construct exists for generating signals within the current context, i.e. generate a signal which is caught by the `handle` statement in whose scope the statement occurs. An example of such a construct would be the `exit` statement in Clu [21]. In Screed, such a construct in not necessary: the actor can explicitly send a message to the appropriate exception handler.

3.3 Supporting Adaptive Dependability

A significant difference between exception handling in Screed and other languages is the use of third-party exception handlers. In languages such as CLU [21], SR [18], and Ada [10], exception handling routines are defined within the scope of the invoking objects. We refer to this approach as two-party exception handling (the invoker and the signaler) and our approach as three-party exception handling (the invoker, the signaler and an independent handler). We have found that two-party exception handling is unsatisfactory for modeling the complex relationships between objects such as those required for adaptively dependable systems.

The key difference between two- and three-party systems is in the potential *knowledge* of a given object. With two-party exception handling, the exception handler, which is also the invoker, may know only of the exceptions caused by its invocations. Therefore, in such a system it is very difficult to develop a global picture of the fault pattern in the system. In a three-party system, such monitoring may be naturally expressed in terms of the exception handler since it may handle exceptions for a large group of objects or even the entire system. Furthermore, an autonomous exception handler may subscribe to any available failure detectors, thereby augmenting the knowledge received through exception signals.

A third-party exception handler may also be designated as an object with special rights. In this manner the system may be safely modified in response to exceptions and failures. Since it is dangerous to allow the arbitrary modification of one actor by another, most two-party systems can express reconfiguration of the system only by mimicking a three-party system, i.e. they must notify a third object with special rights of the exceptions they encounter and this object may then reconfigure the system.

Thus in adaptively dependable systems, the resulting system architectures will look quite similar to Figure 3.4. In this figure, the manager M receives input in the form of exceptions from application objects and notices from the failure detector. Upon determination that the *Primary Node* is unstable, M allocates the *Backup Node* and creates the appropriate objects to replace the *Primary Node*. Note that, in actuality, M is probably a replicated entity to ensure its resilience and availability.

Systems with such architectures may allow the dynamic installation of dependability protocols or may simply support the reconfiguration of several objects in response to exceptions. In either case, the system will have a *manager* with special rights to modify other objects. The manager will act as the exception handler for some group of objects being monitored. Furthermore, the manager may also subscribe to failure-detection services. Upon receiving enough input to determine that some unacceptable condition exists, the manager reconfigures the group over which it has authority.

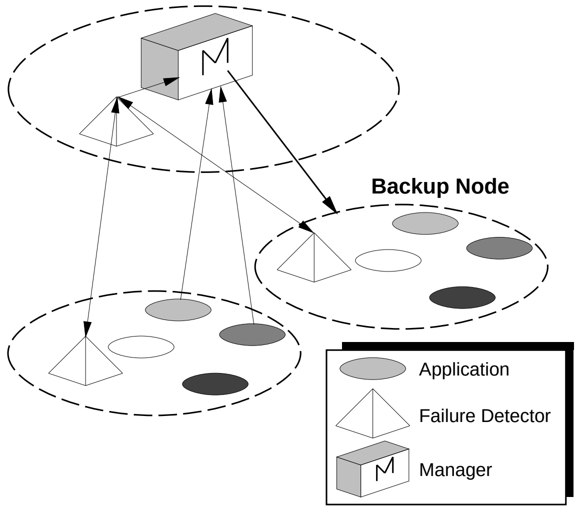

Figure 3.4: A prototypical adaptively dependable system.

3.4 Reconfiguring a Satellite to Preserve Failure Semantics

To illustrate the concepts we described above, consider a distributed system which is operating in a hostile environment. A good example of such a system is a satellite with multiple processors, each with its own memory. Assume that the memory of these processors was developed to never return incorrect data to a *read*. Instead, the memory will detect the error through some checksum algorithm and return an error condition value. This reserved value can then be used to signal an exception and initiate forward error recovery. The specifications for this system state that such memory errors should occur with probability 10^{-8}. Such a system is shown in Figure 3.5. Notice that the *manager* is itself replicated to ensure fault-tolerance in this vital system component.

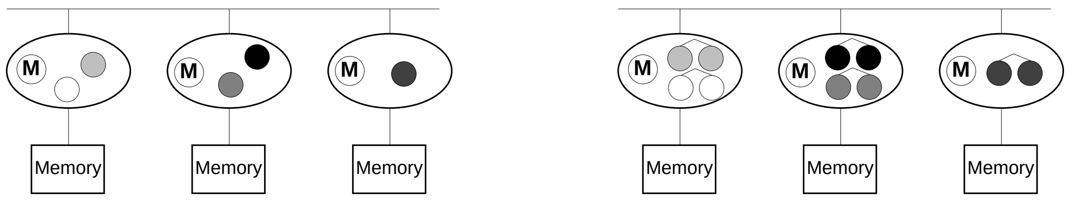

Figure 3.5: Example satallite architecture and result of replication.

Once the system is launched, these memory components seem to operate correctly and the *manager* responsible for memory errors occasionally performs forward error recovery on the objects. However, the rate at which these memory errors are occurring is unacceptably high (10^{-5} faults/read) and system performance degrades significantly due to repeated memory faults and subsequent error recovery. Therefore, the manager installs the replication protocol described in Section 2.3. The resulting system is also shown in Figure 3.5. Since both the original actor and the replica will be reading values from different memory locations, the probability that a memory error will be noticed is now 10^{-10}, well within the specified tolerance. When an exception occurs, the manager will still have to perform some corrections, but the system can keep computing during this time and the error recovery will be simplified due to the existence of the replica. Considering the nature of the faults, the replica may be placed on the same node as the original actor. However, if instead of signalling an exception, nodes crashed when they could not read memory, the replicas would have been placed on different nodes.

Chapter 4

Implementation

Broadway is a platform designed to facilitate experiments with distributed systems. To test our ideas involving adaptively dependable systems, we have implemented the meta-architecture described in Chapter 2 and provided support for exception handling as described in Chapter 3.

In the following sections, we provide an overview of the Broadway system structure and then discuss the details of supporting adaptive dependability. We also will present some performance numbers for Broadway. Although the system was not designed for performance — instead being oriented towards customizability and supporting experiments — we feel that comparing the cost of basic operations provides insight into the use of actor systems.

4.1 The Structure of Broadway

In this section, we describe the structure of the Broadway system and briefly outline its core components. Additional details on the structure of these components may be found in Appendix A.

Broadway's functionality is supplied by a central Dæmon as well as a set of system actors described by a library. The Dæmon is the core of the Broadway system, supporting the scheduling, creation, and communication between actors. Since the Dæmon implements communication, it is here that MAUD is implemented. Other, more complex functionality, is supported though a library of system actors. These actors build upon the basic functionality supplied by the Dæmon to support more complex behavior such as actor migration, exception handling, fault detection, or I/O.

Broadway actors are represented as instances of C++ classes. To provide the interface needed by the Dæmon to manipulate actors, all actor classes must inherit from the class *Actor*. This class maintains a mail queue, handles the transmission of messages, and supports the scheduling of an actor.

4.1.1 Dæmon Structure

The Dæmon was written using C++ using an object-oriented design methodology and, therefore, consists of several well-defined components. These components may be individually re-implemented to allow easy customization of the system. Below we present an outline of the functionality and structure of each of these components.

Scheduler The current scheduling strategy on Broadway is based on a single thread of execution shared by both the Dæmon and all the actors on a node. Such a strategy was chosen for its simplicity. Each actor processes one method invocation (message) in a round-robin manner with the Dæmon performing any necessary functions (such as receiving off-node messages) between method invocations. Methods are, therefore, atomic actions and may not be blocked after execution commences.

Membership Monitor This object keeps track of which nodes are currently running Broadway. When nodes crash, this object is notified. It will then generate a signal to the system actor serving as the failure detector. Furthermore, the Membership Monitor is always queried before a message is sent off node. If a message is being sent to a non-existant node (i.e. the node crashed previously), this object will signal a system actor to generate an exception.

Local Actor Table This table contains a reference to each actor on the local node. Any actor not in this set is considered non-local.

Foreign Actor Table Actor address include the identity of the creating node. Therefore, this table need only store the addresses of those actors which have migrated off their creating node. Such a structure keeps this table small and look-ups inexpensive.

Behavior Table Given a *Behavior* value (i.e. a unique identifier representing a particular actor class), the Behavior Table will provide a constructor for an instance of that class. This table enables off-node creation and migration of actors.

Factory Much in the same manner as the Behavior Table supports the creation of new actors, the Factory supports the creation of primitive type values included in messages. Each primitive type (i.e. those types which may be included in messages) has a unique identifier. When an off-node message has been received, it will be in the form of a character stream. For each value, the first two bytes in the stream identify its type. Given the byte representation of a value, the Factory will then create a new instance of the correct type, initializing it with the data in the character stream. In this manner, Broadway programmers may include their own primitive types by simply registering them with the Factory object.

Platform Since our scheduling is architecture-independent, the only machine-dependent portion of Broadway is inter-node communication. This functionality is encapsulated in the Platform object. The Platform object presents a uniform interface for inter-node communication so that only the implementation of this object need be changed to port Broadway. Currently, Broadway runs on ULTRIX DecStations and SunOS SparcStations. A port to the CM-5 is currently underway.

All of the above components are linked by the Dæmon object. It is through calls to the Dæmon object itself that local actors utilize these other components. For example, an actor may ask the Dæmon to create a new actor. This involves creating a new instance using the Behavior Table and then adding it to the Scheduler.

4.1.2 Library

One of the primary design goals of Broadway was to provide a flexible platform for the development of different concurrent programming languages. A programming language on Broadway must generate C++ code where all actor classes inherit from *Actor*. This is the sole restriction placed on the nature of the programming language. Therefore, to support a large range of potential languages, we have made the structure of the Dæmon as general as possible. However, there are still parts of the system which must be dependent on the language conventions in use.

For example, consider exception handling. There is a specific structure for exception handling in Screed, as described in Chapter 3. This structure involves formats for signal messages, for exceptions, and for handlers which are specific to Screed. Therefore, although the run-time system must be the one to detect most exceptions, it should not generate the exception objects and signal messages itself.

We implement such functionality in terms of a system library. For the exception handling case, the Dæmon is given a set of function calls to make upon detection of specific exceptional conditions. These function calls are to methods in a Screed actor. This Exception actor may then generate objects and messages consistent with Screed formats. The library actors designed for Screed were written primarily in Screed with minor modifications of the resultant C++ code.

Exception Generator As described above, this actor generates the appropriate exception objects and signals when an exceptional condition is discovered by Broadway.

Failure Detector This actor is similar to the Exception Generator. However, since individual actors may "subscribe" to the failure detector, it is a separate entity. The Failure Detector is notified whenever Broadway detects a node crash or an actor crash.

Control This actor is arguable the most important actor on a Broadway node. Control is the actor interface to the Dæmon itself. Control handles off-node requests for actor creation and migration. Control also receives notification from other nodes when an actor has migrated, thereby keeping the *Foreign Actor Table* accurate.

Timer The Timer provides a way for applications to measure performance. Timer provides a stop-watch facility based on wall-clock time. The Timer receives requests and measures its time using the Broadway Dæmon. Current implementations of Broadway call the UNIX `gettimeofday` routine to return the time.

Terminal The Terminal actor allows formatted input and output to the user. The number of Terminal objects in the system is highly dependent on the system configuration. By providing the Terminal as a library actor, the number and configuration of user i/o may be controlled by the language designer.

Most extensions to Broadway should be in terms of library actors. For example, actor migration was added to Broadway by extending the functionality of the Control actor.

4.2 Implementation of MAUD

To support dependability, the MAUD meta-architecture, as described in Chapter 2, was implemented on Broadway. Implementing general reflection involves the use of run-time interpretation of all reflected objects. Since MAUD is a strictly limited meta-architecuture, however, it was possible to implement MAUD efficiently using compiled objects and redirection of messages.

As described in Section 4.1, each actor on Broadway is represented as an instance of a C++ class. Since these actors inherit from the class *Actor*, each has several predefined methods and state variables. We implemented MAUD through the modification of these predefined methods.

Specifically, each actor has a `send` C++ method which will transmit a message, and a `deliver` C++ method which places an incoming message in its mail queue. Each actor also keeps two acquaintance variables — *dispatcher* and *mailq* — representing its meta-level objects. If either of these two objects have not yet been reified, the acquaintance value is `null`. Through operations on these two variables the methods `change_mailq`, `change_dispatcher`, `add_mailq`, and `add_dispatcher` were easily implemented for all actors.

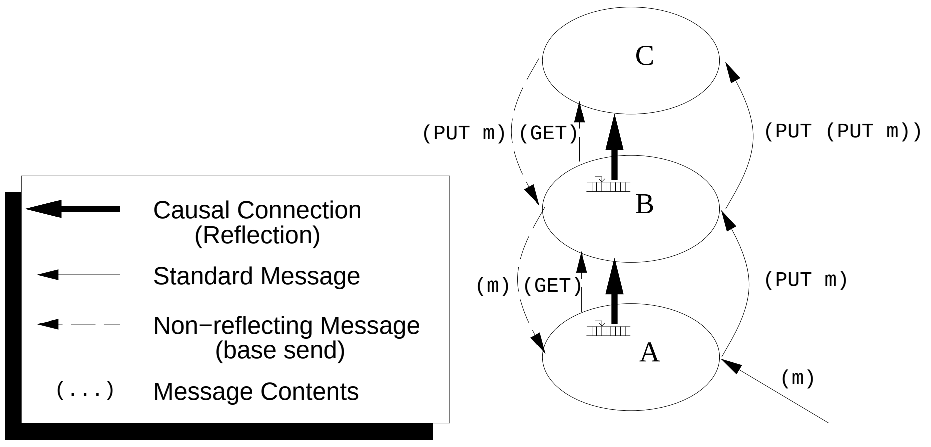

Figure 4.1: Messages sent between meta- and base objects.

To implement the *dispatcher* component of our meta-level, we modified the `send` method of the *Actor* class so that it checks if a reified dispatcher exists before transmission of the message. If such an actor does exist, a new message is constructed which will server as a `transmit` method invocation and has the original message as the sole argument. This message is then sent to the `dispatcher` acquaintance. In this manner, any send operation in a base actor will result in a `transmit` method invocation in its meta-level dispatcher. If the meta-level dispatcher has its own meta-level dispatcher, this meta-meta-object will receive its own `transmit` message if the meta-object transmits a message. In this manner, layered protocols operate correctly.

A similar technique is used to implement meta-level mail queues. Figure 4.1 shows the actual messages resulting when a base object with a two-level mail queue hierarchy receives a message. In our original (unoptimized) implementation, when the `deliver` method is invoked on the actor's C++ representation, the `mailq` acquaintance is checked and, if non-null, the message is sent to the meta-level mail queue. As with the meta-level dispatcher implementation, the original message becomes the only argument to a new message. For the mail queue implementation, this new message will invoke the method `put` at the destination actor. When the meta-level mail queue receives the message, its `deliver` method will, in turn, check to see if it also has a meta-level mail queue. If so, this mail queue will receive its own `put` message with the old `put` message as the sole argument. In this manner, the highest level mail queue — i.e. the meta-level mail queue which has no meta-level mail queue of its own — will be the first to process the message.

As described in Section 2.1, each meta-level mail queue must also have a `get` method. The mail queue should store all messages to be delivered to its base object until it receives a `get` request from the Broadway Dæmon. Broadway will issue a get request when the base object needs another message to process. If the mail queue has no appropriate message to pass onto its base object, it must record that a `get` message was received: no further `get` messages will be sent by the Dæmon until the base actor processes another message. Once synchronization constraints are implemented in Broadway [15], a constraint would prevent the delivery of a `get` message until the meta-level mail queue is prepared for the message (i.e. has messages to deliver).

Unfortunately, the above implementation for mail queues involves a newly received message being passed "up" between mail queues using `put` messages and then "down" using `get` messages. Half of these messages (the `put` messages) may be eliminated using *caching* of the address of the mail queue at the highest meta-level. The address of this mail queue is recorded at the base object as well as the current number of meta-levels. Upon reception of a message at the base object, the appropriate message is created as if it was passed between the meta-level mail queues. This message is then sent directly to the mail queue at the highest meta-level.

Using caching in this manner, however, requires that new mail queues be added *atomically*. Therefore was have also added the methods `begin_protocol` and `end_protocol`. These methods work much like a test-and-set operation, checking to see if another protocol is being installed and, if not, reserving the right for a particular entity to install a protocol (i.e. a mail queue and dispatcher). Furthermore, no messages will be processed while protocols are being installed. Although we have not yet included security functionality in Broadway, it is in these methods that any authorization checks would be made to ensure a trusted entity is installing the protocol.

One last component necessary for successful implementation of MAUD is the ability to send messages which are not processed by meta-objects. For example, a mail queue's response to a `get` message must be sent in this manner or the message will be repeatedly processed by the same meta-level mail queues. The meta-level operations for adding mail queues must also be processed in this manner. To support these messages, Broadway provides a `base_send` operator which marks a message to not be processed by any meta-level objects.

4.3 Implementation of Screed

For the most part, our Screed compiler performs a simple translation from Screed to C++ with calls to the Broadway Dæmon. However, there are two key aspects of Screed that require detailed implementation discussion. First, we will discuss *explicit continuations*. Explicit continuations are used whenever a value must be returned from a message send. Although an explicit continuation is quite similar to a remote procedure call, it does not involve any explicit blocking of a method.

We will then discuss how we implemented the exception handling scheme proposed in Chapter 3. As mentioned in Section 4.1, Broadway itself only provides very primitive exception handling mechanisms in the form of a function call to a local actor. To implement exception

```
foo(int x) {
   var
      int y;
   A.boo(x,y)(int ret) {
       w := x + y + z;
   }
}
```

```
_cont_meth_1(int ret) {
   var
      int y;
      int x;
   x = _state_[0];
   y = _state_[1];
   w := x + y + z;
}
foo(int x) {
   var
      int y;
   A.boo(self,_cont_meth_1,x,y);
   _state_[0] = x;
   _state_[1] = y;
}
```

Figure 4.2: Example transformation of an explicit continuation.

handling for Screed, it was necessary to construct the appropriate system library actors as well as generate the correct code for `handle` and `signal` statements.

4.3.1 Explicit Continuations

An explicit continuation is specified as a message send with an additional parameter list for return values. Currently, this auxiliary list must consist of either zero or one value. However, we plan to add multiple return values in future versions of Screed.

The syntax for the explicit continuation is specified as follows:

```
<actor>.<method>(<arg-list>)([<ret-type><ret-var>]) {
   <expr-list>
}
```

Initially, this command acts as a standard Screed method invocation: a message is sent to *actor* for the invocation of *method* with the specified argument list. The invoked actor is then expected to return a value of type *ret-type*. The type of this return value is type-checked dynamically by Broadway when the continuation is invoked. The continuation body will execute after this value has been returned with the returned value being bound to the variable *ret-var*.

However, the rest of the enclosing method will be executed immediately after the initial message is sent. Therefore, the context in which the continuation will execute will have all variables holding the value they had at the termination of the enclosing method. In the case of state-variables, these values may have changed even more due to other pending messages which the actor processes before the continuation. With the eventual introduction of synchronization constraints [15] to the language, the programmer will be better able to specify which methods may execute before completion of the continuation.

The implementation of explicit continuations was fairly straight forward. The continuation body itself is converted into a new method. This new method has, as its only parameter, the return value parameter specified. It also has local variables corresponding to any local variables in the enclosing Screed method as well as the parameters of the local Screed method. The continuation body is then processed in this manner to allow the nesting of explicit continuations.

The explicit continuation is then converted to a standard asynchronous message send whose first two parameters are the address of the sending object and the name of the return method. To preserve consistency, the Screed compiler put these two additional parameters on all messages; when there is no return method, null values are supplied. Broadway will automatically drop any messages to a null actor.

Before the enclosing method ends, all local values (including the method parameters) are saved in a predefined array called `_state_`. `_state_` is then unpacked into the appropriate local variables. An example of this transformation is shown in Figure 4.2.

One advantage of explicit continuations over RPC communication is that no blocking of the method is necessary. This also prevents deadlock in cases such as an object sending an RPC to itself. Although such deadlock would still be possible though the naive use of synchronization constraints, these constraints are completely under the control of the programmer. The programmer could exploit semantical information concerning the object to minimized the synchronization whereas a compiler would always have to be conservative.

4.3.2 Implementation of Exception Handling

The exception handling tools provided by Broadway were designed to be generic and simple. On Broadway, each message holds the address of an exception handler to which all signal notification must be sent. To support system exceptions, Broadway calls a method in the Exception system actor. It is then the job of this system actor to generate an exception as detailed by the programming language.

The details of the Screed exception handling syntax were given in Section 3.2. We begin with a discussion of how the `handle` statement was implemented and then discuss the `signal` statement and our Exception system actor.

4.3.2.1 Handle Statements

Most of the complexity in implementing the exception handling mechanism of Screed was involved with the `handle` statement. Since exceptions are rare events, it was decided that a minimal amount of exception handling information should be included with each Broadway message. Therefore, Broadway limits each message to one exception handling address. Screed must construct a single exception handler which will route the exception to the appropriate handler specified in the Screed program. Although this involves extra messages, the goal of the implementation was to make non-exception generating messages efficient even at the expense of signal efficiency.

Screed uses the predefined class *Directory* to route exception signals. The Directory is initialized with two array's and the size of these arrays. The Screed code for this class is shown in Figure 4.3. Two array's are used since structures have not yet been implemented for Screed. A new Directory object is created for each new handle statement executed: there is a one-to-one correspondence between Directory objects and handler scopes. Since exception handling is asynchronous, the `handler` environment which existed when the message was sent must be preserved even while the invoking actor continues to process other messages.

The address of the Directory object is then included with each message in the handle scope. To implement nested handler scopes, each Directory object is also initialized with a reference to

```
class Directory {
   var
      method exceptions[MAXESIZE];
      actor handlers[MAXESIZE];
      int num;
      actor up;

   init(int n, method ex[], actor h[], actor u) {
      var int i;

      num := n;
      up := u;
      exceptions := ex;
      handlers := h;
   }

   exception(actor ex) {
      ex.name()(method n) {
         var
            int i;
            int done;

         done := 0;
         i := 0;
         while (i != num & !done)
            if (exceptions[i] == n)
               done := 1;
          if (i != num)
            handlers[i].n(ex);
          else up.exception(ex);
      }
   }
}
```

Figure 4.3: Code for the *Directory* class.

its parent scope. If a specific exception is not registered with a Directory, it passes the exception to its parent. To correctly preserve this scoping, each object now maintains a handler stack: when a new scope is entered, the new Directory is created, initialized, and its address pushed onto the stack; when the scope terminates, the address is popped off the stack. The top level environment is initialized with the address of *Default-Handler*: the global exception handler. The value *current_handler*, which is a state variable defined in the Broadway *Actor* class, is always re-initialized to the top stack value whenever the stack is modified. Figure 4.4 shows a handle statement and its transformation into C++ for Broadway.

4.3.2.2 Signals & the Exception System Actor

Signal statements are a simple translation into Broadway C++. The *Actor* class supports a `signal` method which is identical to `send` except that no destination is specified: the exception handler address of the current message is used instead. Therefore a Screed `signal` is translated to the creation of the exception object and a standard message send.

```
handle (arithmetic with myhandler) {
   B.divide(D)(actor E) {
      myNum := res;
   }
}
```

```
old_handler := current_handler;
current_handler := new Directory;
_handle_stack.push(current_handler);

ex[0] := arithmetic;
h[0] := myhandler;
current_handler.init(1,ex,h,old_handler);

B.divide(D)(actor E) {
   myNum := res;
}

_handle_stack.pop();
current_handler := _handle_stack.top();
```

Figure 4.4: Transformation of a `handle` statement into C++.

The Exception system actor is also quite simple. At start-up, one Exception actor is created on each node; this actor may not migrate. It contains a method for each system exception and Broadway is initialized with the physical address of the functions implementing these methods. These methods are then called directly by Broadway which provides the information detailed on page 17. Note that this always includes the message which caused the exception. The Exception actor creates an instance of the appropriate exception object and sends a signal to the handler specified in original message.

4.4 Performance

In this section, we present several performance numbers for Broadway. Broadway has been designed as an easily customizable test-bed for new concepts in distributed computing. Therefore, the design emphasis has not been on optimization. Even so, we felt presenting performance numbers would give a valuable insight into the use of actor systems.

Specifically, we compare several values for the UNIX implementation of Broadway. In particular, we compare communication costs and actor creation costs for both local and remote operations. The experiments were performed on a group of Sun 4 workstations at the University of Illinois.

	Message (Short)	Message (Long)	Creation
Local	$348\mu s$	$378\mu s$	$289\mu s$
Remote	$1620\mu s$	$5140\mu s$	$6570\mu s$

Figure 4.5: Performance times for various Broadway operations.

Figure 4.5 presents the times for these operations on Broadway. Local operations are those occurring with actors on a single node; remote operations involve inter-node communication. The message time is the time taken to send a message in one direction. Short messages are sent with a single integer parameter. Long messages contain an additional array of 100 integers.

Since local arrays are always passed by reference, the difference in the local case in minimal. However, the difference for the remote case is significant.

Actor creation is quite simple in the local case, involving only the allocation and initialization of a C++ object. This is important for actor languages on medium or low granularity architectures: many more actors will be created than the number of processors. For off-node creation, at least three remote messages must be used. Therefore, creating a off-node actor is an expensive operation. However, multiple off-node actor creations may be chained which will significantly reduce the overhead involved. We used this technique with 1000 remote actor creations which completed in $3.12s$. Therefore, chained off-node creation is, on average, slightly less than twice as expensive as an off-node message send.

The limiting factor on the off-node operations was the network protocol itself: socket communication over ethernet is quite slow. We expect that on a higher performance architecture, such as the Thinking Machines Corporation's CM-5, our "remote" results will be significantly better.

Chapter 5

Conclusion

5.1 Summary

In this thesis, we have described a methodology for the development of adaptively dependable systems. Adaptively dependable systems may function over a long duration despite a changing execution environment. Whether the changes are due to a variance in the components comprising the system or to a change in the physical environment in which the component operates, the use of dynamic protocol installation combined with exception handling allows fault tolerance to be guaranteed by the system.

Dynamic protocol installation is enabled through the use of a meta-level architecture. Our meta-level architecture, MAUD, allows the customization of an object's communication behavior on a per-object basis. By describing protocols in terms of modifications to the communication behavior, protocols may be dynamically installed on objects as necessary. Furthermore, if a protocol is no longer required, it may be removed. Through the use of caching and atomic protocol installation, the meta-level description of protocols may be implemented with a minimal cost in performance.

We also support composition of protocols. Provided there are no inherent semantic conflicts between two protocols and both protocols are implemented using MAUD, these two protocols may then be composed without foreknowledge that a composition may occur. In this manner, protocols may be constructed in a modular fashion and later combined to provide the desired level of fault-tolerance.

To provide adaptive dependability, we combine dynamic protocol installation with exception handling. We make extensive use of *third-party* exception handlers which are shared between multiple objects. Since these handlers have privileges to modify meta-level objects, they are termed *managers*. A single manager will be informed of all exceptions related to a particular problem. The knowledge managers obtain as exception handlers may be augmented through subscription to failure-detection services. In this manner, the manager will have all information necessary for a correct diagnosis of fault patterns.

The concepts described in this chapter have been implemented on Broadway — our actor run-time system — and are accessed through the language Screed. Screed provides exception handling constructs which support managers and provides access to the meta-level architecture implemented in Broadway.

5.2 Future Work

5.2.1 Consistent Installation of Protocols

When installing a protocol on multiple objects, it is necessary to ensure consistency. Different protocols may have different rules for defining consistent installation. The best tool for coordinating such operations is a multi-object constraint.

Specifically, *synchronizers* [16] may be used to specify customized constraints for each protocol or even a subset of objects in each protocol. Synchronizers are expressed in terms of events at the coordinated objects. These events may be restricted in several different ways including *mutual exclusion*, *atomicity*, or *triggering*.

Synchronizers are currently being added to Broadway and Screed. When complete, protocols may be written in terms of MAUD and then have the rules for their consistent installation expressed in terms of Synchronizers. Synchronizers are independent of the classes and objects being coordinated. Therefore, a single installation policy may be reused with multiple protocols.

5.2.2 Protocol Languages

The techniques proposed in this paper simplify the construction of dependability protocols by separating application code from protocol code. Furthermore, transparency is preserved by constructing protocols as operations on messages. However, protocols written using MAUD may still be complex. This can be especially true when a protocol must follow the rules outlined in Section 2.4 for composition.

We are currently investigating techniques for specifying protocols at a higher level. Again, the goal is to specify protocols in terms of operations on messages, but to also express communication as being between arbitrary clients and a fault-tolerant service. The concepts of delay, message ordering, and failure should also be incorporated at a high level.

These higher-level protocol specifications may then be "compiled down" into a meta-level specification such as MAUD.

APPENDIX A

Broadway

This appendix describes, in detail, the design and implementation of Broadway. Specifically, this section is designed to serve as a guide to both programmers designing languages for Broadway as well as those who are modifying Broadway itself.

We begin with a discussion of primitive data types — those data types which may be message values — and describe how programmers may add their own primitive types. We then explain the internal representation of actors and how inheritance is used to add language-based functionality to the basic actor objects.

Once the pertinent data structures have been explained, we describe the Dæmon object and the internal objects from which it is composed. These descriptions are aimed at those who wish to customize Broadway itself. We describe the interface of each object as well as its current implementation and interactions with other objects.

A.1 Data Structures

Every implementation of Broadway uses a set of primitive data types. By primitive we mean that these types may exist as parameters in messages. There are currently eight such types: `Integer`, `Real`, `String`, `Address`, `Method`, `Behavior`, `Msg` and `Array`. However, to support customization and the design of new actor languages, Broadway supports the development of new primitive data structures. This support allows the inclusion of instances of these primitive data types in messages sent off-node. We begin with a discussion of the built-in data structures, discuss the structure of all primitive types, and then explain how new data structures may be incorporated into Broadway.

A.1.1 A Basic Set of Data Structures

The default set of data types are those needed with almost any actor language. `Integer` and `String` are self-explanatory; `Real` are double precision floating-point numbers; `Address`, `Behavior`, and `Method` are used to reference actors, actor classes, and actor methods, respectively. Each element of an `Array` may be any of these data-types. Multi-dimentional array's are realized by having an `Array` whose elements are of type `Array`. To facilitate the development of meta-level actors, it was necessary to make `Msg` a first-class values.

The interfaces to these data types is fairly straightforward. `Integer` can be interchanged with the C++ `int` through type-casts or even assignment (=). The same applies to `String` with `char *` and `Real` with `double`. An `Address` is only created through the use of the build-in routines `create` and `create_off`, i.e. an address is never directly created by the user. `Behavior` and `Method` are initialized at creation using a constructor which takes one integer. It is up to the language to preserve the uniqueness of behaviors and methods: no naming service is currently provided with Broadway. `Address`, `Behavior`, and `Method` may be assigned or compared for (in)equality to other values of the same type.

An `Array` may be subscripted, as with normal C++ arrays. Each element of an `Array` is actually a pointer to a value. Therefore, memory must be allocated for an array element before a value may be assigned to it. Assignment of an entire `Array` is also possible. However, the assignment operation is expensive as it copies each element of the array. `Array` also has a `resize()` operator which preserves as many original values as possible.

The `Array` constructor may be supplied with no arguments — resulting in a `Array` of size 0 — one argument specifying the size, or two arguments specifying the size of the `Array` and type of each element. When the array element's type is specified, Broadway allocates an element of the correct type off the heap (using the C++ `new` command). `Array`'s automatically check for out-of-range element accesses.

`Msg` objects consist of a destination address, method to invoke at the destination, and an array of contents. Messages may be directly subscripted to reference these contents. To assign a new value to a message, the `add_value` method should be used. This procedure places the new value in the next available field and increments a counter representing the number of parameters.

The `Msg` object constructor takes a single value: the maximum number of parameters the message can take. If this value is not specified, it defaults to 0. The message objects also support a `resize` operator.

A.1.2 Data Type Characteristics

All primitive data types must inherit from the class `Value`. The `Value` class is an abstract class which enforces the definition of several methods needed for message passing. These methods are:

`short myclass()` Return the integer value representing the object's type. This value must be unique to this subclass of `Value` and must be the first item "packed" when the converting the data structure to a character stream.

`int pack(char *buf)` Pack the data structure into the character buffer. The number of bytes used to pack the object is returned.

`char *unpack(char *buf)` Unpack a data structure of this type from the character buffer. A pointer to the start of the next object packed into the character buffer is returned.

`void print()` Print the value of the data structure: this method is used only when debugging.

Classes which define the above methods provide the support necessary to convert objects of this class into byte streams and back. This support is necessary to include an object of this class in a message being transmitted to an actor on another physical node. Although this support is necessary, it is not sufficient to support a primitive type.

Many operations in Broadway must operate on arbitrary primitive values in the form of a `Value *`. In particular, array and message subscripting return a pointer to a value. To allow the correct conversion and type checking of `Value *` to primitive types, type converter functions are defined. These functions have definitions of the form:

```
TYPE &ValToTYPE(Value *)
```

where `TYPE` is the primitive type's name. For example, to convert a `Value *` to an `Integer`, the function `ValToInteger` is called. Since a reference to `TYPE` is returned from these functions, they do not result in the value being copied.

A.1.3 Designing Custom Data Structures

As mentioned previously, it is possible for a programmer to define new primitive data types. The primary requirement for defining a new type is that the new data type inherit from `Value` and define all the methods required by this abstract class.

Two macros must then be used to enable use of the data type in off-node messages. These macros register the type with the Dæmon's Factory to allow creation of new objects from character streams. They also define the `ValToTYPE` function mentioned above.

MAKETYPE(type, id) This macro must appear at the global level. `type` is the name of the type and `id` is the integer value which will be returned by the `myclass` method of the type. `MAKETYPE` defines the `ValToTYPE` operator as well as a function for creation of a new instance of the type.

REGISTERTYPE(type, id) This macro must be in code which is executed before the type is ever used. `type` is the name of the type being registered and `id` is the value which will be returned by the `myclass` method of the type. `REGISTERTYPE` informs the Factory that when the next value in a character stream begins with `id`, it will be a value of type `type`.

For the default types, these macros are used in the file `typelib.cc`. This file serves as an example of how these macros may be used.

A.2 Structure of Actor Class C++ Representations

All actors in Broadway must be represented by a C++ class which inherits from the class `AClass`. For this reason, the file `AClass.h` must be included in all files which declare actor classes. The state of an actor should be represented as private items of a C++ class; the methods should be public items of the C++ class.

In addition to defining the methods specified for the actor, a lookup function must also be provided. This function should take the form:

```
void lookup(Method& meth, Array& v, int param)
```

The definition of `lookup` must be public. The lookup method will be called by the system when there is a message to be processed. `lookup` should call the correct C++ method in the actor class as specified by the name `meth`. If a matching method cannot be found, the parent class's lookup function should be called. The number of parameters is provided in `param` to prevent the invocation of a method will invalid parameters.

The array `v` contains the parameters to the method. Each element of `v` used as a parameter should be typechecked and converted to the correct type. Using the appropriate `ValtoTYPE`

```
void Database::lookup(Method& m, Array &v, int params)
{
  if (m == query_n && params == 3)
    query(ValToInteger(v[0]),
  ValToAddress(v[1]),
  ValToMethod(v[2]));
  else if (m == update_n && params == 4)
    update(ValToInteger(v[0]),
           ValToInteger(v[1]),
           ValToAddress(v[2]),
           ValToMethod(v[3]));
  else if (m == read_n && params == 3)
    read(ValToInteger(v[0]),
         ValToAddress(v[1]),
         ValToMethod(v[2]));
  else AClass::lookup(m,v,params);
}
```

Figure A.1: A sample lookup function.

function guarantees a correct conversion, as mentioned in Section A.1. A lookup function for a database actor is shown in figure A.1.

Interface to the Dæmon is provided through routines inherited from `AClass`. There are five such routines.

`Address create(Behavior &)` The daemon is called to look up the behavior, create an actor of that class, and return the address of the new actor.

`Address *create_off(int node_num, Behavior)` Functions as with create, except that an actor of the specified behavior is created on the node with id *node_num*. This method returns the address of the newly created actor.

`void send(Address *dest, Method *meth, arg1, arg2..., 0)` Takes a variable number of parameters terminated by a 0. These parameters are packed into a message. Both the destination and the method must be provided. All arguments must be *pointers* to values due to the nature of the C++ *varargs* package.

`void dispatcher(Address disp)` A meta-level operation which installs the actor with address `disp` as the dispatcher of this actor.

`void mailq(Address mailq)` A meta-level operation which installs the actor with address `mailq` as the mail queue of this actor.

Note that these routines may only be used from within an actor's method: only actors may create other actors, send messages, or modify their meta-level.

Each new behavior must also be registered with the Dæmon. This is done through the macro `DEF_BEH`. This macro must be provided with the `Behavior` value representing the behavior, the class name, and the path for the object file defining the class (for later use with dynamic loading).

For example, a class `User`, which is represented by `Behavior` "user_b" and may be found in "/obj/user.o" would be defined as.

```
DEFBEH(user_b, User,"/obj/user.o");
```

Note that the use of this macro should be at the *global* level, not in the setup function. In general, it is best to place these statements at the end of the file.

A.3 Actor Objects

As detailed above, each actor on Broadway has a corresponding instance of a C++ class. The instances always inherit from the class *AClass*, which provides many of the necessary built-in methods. However, there is a set of methods and fields associated with each object which was not discussed in Section A.2. These are the methods and fields required by other objects in Broadway itself, regardless of the language being used. These methods and fields are defined in the class *Actor*. Language-specific functionality is added in the class *AClass* which inherits from the class *Actor* thereby guaranteeing that all actors satisfy the requirements for Broadway as well as the programming language being designed.

The methods in *AClass* are primarily helper methods to interface the actor's code with the Dæmon and provide functionality to assist the language designer. The methods and fields in *Actor*, however, assist in the scheduling and manipulation of the actor by the Dæmon. Therefore, these methods and fields, unlike those in *AClass*, are necessary for actors to exist on Broadway. A different language may be better implemented with a class other than *AClass*. In fact, the authors encourage language designers to develop their own custom root class. Any such new class, however, must always inherit from *Actor* to allow these objects to be scheduled, receive messages, etc.

The methods of the class *Actor* are oriented towards the scheduling of the actors and actor communication. To support communication, the class *Actor* provides the methods `send` and `deliver`. The `send` method transmits a new message. This transmission involves the location of the destination actor and the use of the Dæmon if off-node communication is necessary. If the message is local, the parameters are copied and the message is placed in the destination's mail queue. The `deliver` method is invoked to place a new message in the actor's mail queue. This method may be invoked by either a local actor performing a `send` or the Dæmon upon reception of an off-node message. These two methods also support Broadway's meta-level architecture by routing messages to reified *dispatchers* and *mail queues* as appropriate (see Section 2.1).

To support scheduling, an actor may be in one of three states: *active*, *pending*, or *buffer*. An actor in *active* state behaves as described above: this is the default state for all actors. An actor which has an empty mail queue is *pending* and, for obvious performance reasons, is not invoked by the scheduler until a new message is received. The methods `pend` marks the actor as in the pending state and the method `is_pend` returns `true` if the actor is pending.

The *buffer* state is used to support migration and off-node creation of actors. When an actor is in the *buffer* state, it may receive messages but will never be scheduled to process a message. Such an actor serves solely as an address to which messages may be delivered. An actor is marked as being in the buffer state by the method `only_buffer` and the method `is_buffer` returns `true` if the actor is a buffer.

An actor is placed in the *buffer* state when it is the process of moving from one node to another. The actor at the old address is made into a *buffer* to record any messages which are received during migration. If this were not done, the actor created at the migration destination could have an inconsistent state from the original actor. Instead, the *buffer* actor holds all messages received during migration. When migration is completed, the `dump_msg` method is invoked and all messages are sent to the new location. The method `dump_msg` forces all messages in the actor's mail queue to be resent. Assuming the local address tables have been updated, the messages will arrive at the new location.

A.4 The Dæmon

The Dæmon object is the heart of the Broadway system, serving as the "glue" between objects such as the Platform, Factory, or Scheduler. It is through the Dæmon object that actors in Broadway access system commands and data structures. The Dæmon object is also the only object to access the Platform object and, therefore, provides a level of abstraction for actors needing to communicate with off-node actors.

Each Dæmon is responsible for the scheduling and maintenance of a particular set of actors. This set of actors is considered the "local" set. Therefore, each Dæmon object defines a logical node in the system. In current implementations of Broadway, there is a one-to-one correspondence between Dæmon objects and physical processors. However, it is not necessary to adhere to this organization and, provided that the Platform object can send messages to other processes on the same physical node, multiple logical nodes may be located on a single physical node. Furthermore, in the case of shared-memory multi-processors, it would even be possible for several physical nodes to be contained in one logical node: one Dæmon object monitors the actors whose threads are distributed over several physical processors.

Any system configuration may be realized in terms of modifications to the Dæmon class and the Platform class. Therefore, for any programmer of the system, the actual system configuration is transparent: each Dæmon object is assumed to have a unique integer identifier which, as with actor addresses, is location independent. For purposes of actor migration, communication with the Dæmon, etc. only this identifier need be known.

Each Actor object has a pointer to the local Dæmon. Through this reference, local actors are able to access system commands and structures. When the node has been initialized and is ready to begin processing, the main Broadway routine calls the Dæmon's `run` method. This infinite loop continually calls the scheduler to process the next actor.

The `send_off` method takes a message and transmits it to an off-node location. The location is specified in the message structure. The Dæmon will also periodically be given permission by the scheduler to process off-node messages it has received. The Dæmon calls `read_mail()` between actor method invocations to read its own mail. In the current implementation of the Dæmon, it will simply call its own private method `receive_off` to process off-node messages.

Actors may be created through the use of the `create` method. This method takes a pointer to an object of type Actor (or any sub-class of Actor), adds the actor to the Scheduler object, and records the actor as a local actor. This routine is often called by objects of the Actor or AClass classes which provide higher level interfaces to `create`.

The `make_buffer` method takes the address of a local actor and modifies that actor to be a message *buffer* (see Section A.3). As a buffer, an actor may receive messages, but is never scheduled and, therefore, never processes messages. Buffers are primarily used when an actor is migrating.

The Dæmon maintains two tables of actors: *Local* and *Foreign*. The Local table contains all actors on the local node, even if they are currently not in an *active* state (see Section A.3). The Foreign table contains a list of all actors who are no longer local to the node where they were created. If an actor is at its creator, then its location may be determined using the `creator` method of any `Address` referencing the actor. Otherwise, the Foreign table gives the proper address.

The Dæmon also maintains a catalog of behaviors: *Behaviors*. This catalog allows the use of first class behaviors and off-node creation of actors. Each entry in the catalog consists of a `Behavior` value (see Section A.1), a pointer to a creation function, and a UNIX path to the object file containing the behavior. This last field is currently unused, but was included in anticipation of a dynamic code loading facility being added to Broadway in the future.

Below we describe the actor interface to the Dæmon. The other components of the Dæmon — Scheduler, Factory, Membership Monitor, and Handler are described in detail in their own sections.

As mentioned above, a pointer to the local Dæmon object is a field in each Actor object. Therefore, local actors may directly access the Dæmon through function calls. For remote accesses, however, special actors exists at each Dæmon which provide system-level operations to off-node actors using the standard message-passing interface.

The most important of these is an instance of the *Control* behavior. This actor has several methods which support off-node access to local Dæmon routines controlling actor creation and migration. This actor is always the first created by any Dæmon so will have the actor address $<$*daemon_id,* 0 $>$. The method `create` at this actor takes an address, a behavior identifier, and an array representing the state of the actor to create, as well as the identity of the Dæmon requesting the creation. The receiving *Control* actor will then manipulate its Dæmon to create an instance of the correct behavior and initialize its state and address. The *Control* will then send a `creation_finished` message to the *Control* actor of the requesting Dæmon object.

The `creation_finished` method of the *Control* actor confirms the completion of an off-node creation. It allows the Dæmon creating the actor to update its actor address tables and to forward any messages waiting for the newly created actor.

A.5 Scheduler

As the name implies, the Scheduler is the object in Broadway which schedules actor method-invocations. Actors may be added and removed from the Scheduler by the Dæmon through the use of the Scheduler's `add` and `remove` methods. The method `donext` allows the Scheduler to execute the next actor method.

In current UNIX versions of Broadway, each node consists of a single thread which must be shared by all actors and the Dæmon. The Dæmon must invoke the Scheduler's `donext` method each time it is ready to process another actor message. In between calls to the Scheduler, the Dæmon performs any necessary processing of its own, such as the delivery of off-node messages.

Such a scheduling scheme was chosen because of its simplicity and efficiency. In particular, since each node has only one thread, there was no need to use any locks or semaphores in Broadway. However, this scheme also has problems. In particular, if any actor invokes a method with an infinite loop, that actor will halt computation on the entire node.

Therefore, it may be desirable to create a multi-threaded system. With the use of a thread package, such an implementation could be achieved through modification of the Scheduler's implementation. With a one-thread per actor system, the `donext` method would simply initiate the Scheduler's actions and the `add` and `remove` would be used to control which actors were computing.

A.6 Type Factory

As mentioned in Section A.1, it is possible to add new primitive data types to the Broadway environment. The difficulty with these new additions is enabling their creation upon reception of an off-node message. It is the purpose of the Dæmon's Factory object to enable this creation.

Each message parameter in an off-node message starts with a two byte header describing its type. This header must be unique to a specific type. It is also preferred that these identifiers are issued in numeric order, i.e. each new data type is issued the least unused header value.

The Factory holds an array in which the i^{th} element is a pointer to a creation function for the type with unique header i. Given this convention, construction of the type factory is simple. When registering a new data type, the Factory's `set` method is invoked with the type header and the creation function. The function is then recorded in the Factory's table.

When a value in a character stream must be unpacked, its header is passed to the Factory by the method `make`. `make` looks up the creation function, calls it, and returns the new `Value *`. The entity calling the factory may then call `unpack` on the `Value *` with the character stream. Since `unpack` is a virtual function, it will initialize the new data structure correctly from the character stream.

A.7 Membership Monitor & Handler

Membership Monitor and Handler together implement the majority of system exceptions on Broadway. The Membership Monitor maintains a list of all nodes which are currently active. In the current UNIX versions of Broadway, all nodes must be active before any computation begins, but may become inactive if a node crashes. The Membership Monitor has a `query` method which takes an integer and an optional pointer to a message. `query` returns `true` if the node represented by the integer is active, `false` otherwise. If a message is supplied to `query` and the queried node is inactive, the Membership Monitor will call the handler with the offending message with the methods `add` and `remove`. Both of these methods take an integer identifying the node whose status is being modified. In some dynamic systems, the number of nodes in the system may increase. Therefore, the Membership Monitor provides a `resize` method which allows the maximum number of nodes to be increased.

The Handler is notified whenever Broadway must generate an exception. The `set_crash` and `set_bad_node` methods take a pointer to an Actor object and a reference to the method in the Actor which will be invoked when an exception of that type must be generated. This Actor object is the *Exception* actor in the set of system actors

The methods `crash` and `bad_node` then generate the exceptions. They invoke the method on the Exception actor which was set earlier. This invocation takes the form of a function call since message formats may be specific to a single language. For example, Screed actors expect the first two parameters in any message to be a continuation address and method. By calling the methods in the Exception actor directly, language-specific details are factored out of Broadway and placed in the language's system library.

A.8 The Platform

The Platform class is designed to be the one class which must be modified when Broadway is ported from one operating system to another[1] The platform object is responsible for all O/S based operations, specifically, those involved with interprocess communication.

The most important methods in the Platform class are the `transmit` and `receive` methods. The `transmit` method takes a buffer, the size of the buffer, and the node id to which the buffer should be transmitted. The `transmit` destination node should *not* be the local node. The `receive` method returns a list of buffers which are the messages received since the previous invocation of this method.

The Platform must also provide some basic operations for the management of message buffers. The method `new_buffer` returns a new buffer (of type `char *`) into which a message to be *transmitted* may be packed. The Dæmon will call `new_buffer` to provide a buffer for each new message. The Dæmon will use this buffer as an argument to `transmit`. It is then the duty of the platform object to determine when this buffer may be freed. This decision is based solely on the semantics of the communication model used by the operating system. Note that in the UNIX implementation, this same buffer is continually reused since sockets always copy the message before transmission.

Since the Platform object allocates buffers for incoming messages, the method `free_buflist` is used to inform the Platform that these buffers are no longer needed by the rest of the system. The Dæmon will call `free_buflist` once it has unpacked the messages in the buffers returned by `receive`.

All initialization of the new platform must be done in the constructor of Platform. In the case of UNIX, when the constructor is called all socket connection are made. The destructor closes these connections when Broadway is terminated.

The only other two methods guaranteed by the Platform object assist the Dæmon in understanding the network configuration. The method `max_nodes` simply returns the number of nodes (including the local node) in the Broadway network. The method `get_local_id` returns the particular node number of the local node.

[1]The only exception would be the redesign of the Scheduler to use a thread package. In such a case, the Scheduler class will obviously have to be rewritten as part of the port.

APPENDIX B

Screed

Screed is a prototypical actor language which runs on the actor platform Broadway. Besides serving to simplify the construction of actor applications on Broadway, Screed also serves as an example of how actor languages may be constructed for Broadway. Screed is by no means a complete language, but provides the basic functionality needed in a object-oriented programming language along with several advanced constructs pertinent to the author's research interests. In the following sections we describe the syntax and semantics for Screed.

B.1 Objects in Screed

Screed is an object-oriented language. Since Screed is an actor language, each object is an autonomous computing component which executes methods concurrently with every other object. Objects communicate through method invocations. These method invocations are represented in terms of asynchronous message passing. Although messages are guaranteed to be delivered (in a non-faulty environment), message ordering is not necessarily preserved; the programmer should consider this when constructing programs.

Each Screed object is a collection of state variables and methods. Since objects may communicate only through method invocation, all state variables are, by default, private variables and may not be accessed by other objects. Currently all methods may be accessed by any other object. Each object has a unique, location-independent address. Addresses are first class values in Screed.

B.2 Syntax

In this section we present the basic syntax of Screed *Version 1.0*. Note that this syntax may change in future versions of Screed. Actor classes are declared in Screed with the **class** statement. The **class** statement takes the name of the new class and a possible parent class from which it inherits. State variables and methods may then follow. As in C and C++, { and } are used to open and close statements. The syntax of the **class** statement is as follows:

```
class <class-name> [: <parent-class-name> ] {
   [ var <declaration-list> ]
   <method-declarations>
}
```

Notice that the declaration of a parent class is optional. All classes in Screed inherit certain methods and state variables from a root class. Not specifying a parent class will default the inheritance from the root class.

A declaration list is simply a set of type and variable name pairs. In each pair the type name is followed by the variable name and each pair is followed by a semicolon. The legal type names are discussed in Section B.3.

A method declaration consists of a method name, a list of formal parameters, and a method body. In the method body, an optional declaration of local variables is followed by a list of expressions. Formally, a method declaration may be described as:

```
<method-name>(<parameter-list>) {
    [ var <declaration-list> ]
    <expression-list>
}
```

The method name is any legal symbol and must be unique only within the set of methods of that class. The parameter list is, again, a list of type-name, variable-name pairs. In this case the pairs are separated by commas; no comma follows the last pair. The declaration list at the head of the method body is identical to the declaration list of state variables in the class definition and the expression list consists of any number of legal statements.

B.2.1 Statements

In this section the basic legal statements which comprise Screed methods are described. Most of the syntax will resemble C or Pascal.

Assignment State, local, and parameter values may all be assigned to. Note, however, that assignment to a parameter value has little effect since all method invocations are call-by-value. An assignment has the following syntax:

```
<lvalue> := <expression>;
```

As discussed in Section B.3, the *lvalue* may be either a variable or an element in an array. The *expression* may be any type-safe combination of Screed binary and unary operators.

If – Else The **if** statement in Screed is identical to that of C. The conditional expression must be of type *int*. The **else** clause is optional, but is always associated with the most recent **if**.

```
if (<cond-expr>) {
    <true-expr-list>
} [ else {
    <false-expr-list>
} ]
```

As with C, the true and false expression lists do not require enclosing braces if they contain only one expression.

While As with the **if** statement, the **while** statement in Screed is identical to that of C. Again, the conditional must be of type *int*.

```
while (<true-expr>) {
    <expr-list>
}
```

For The *for* statement in Screed is not as flexible as in C and more closely mirrors that of Pascal. Because incorrectly written **while** loops may never terminate and, therefore, may lead to the halting of a node, we wanted to provide a loop statement which is guaranteed to be safe. The **for** statement simply increments a pre-declared counter variable. A **step** statement will be added in a later version. The current syntax is:

```
for (<counter> := <start-value> to <end-value>) {
    <expr-list>
}
```

Method Invocation Communication is simple in Screed and the syntax is common to many object-oriented languages. The main difference is that, since *method* is a primitive type, a variable may represent the method to be invoked. In a case where a method name and a method variable share the same name, the method variable will take precedence. The syntax for a method invocation is:

```
<actor-address>.<method>(<argument-list>);
```

The argument list is a set of values, separated by commas. These values may be of any type, but they will be dynamically type-checked by the receiving actor.

Explicit Continuations Besides the regular method invocation, it is also possible in Screed to specify a method invocation with an explicit continuation. This explicit continuation may take a return value from the method invocation and the continuation has access to all variables in the scope in which it was declared.

```
<actor>.<method>(<arg-list>)([<ret-type><ret-var>]) {
    <expr-list>
}
```

The continuation body will execute after the actor receiving the initial message has finished executing the appropriate method. However, the rest of the enclosing method will be executed immediately after the initial message is sent. Therefore, the context in which the continuation will execute will have all variables holding the value they had at the of the enclosing method. In the case of state-variables, these values may have changed even more due to other pending messages which the actor processes before the continuation. With the eventual introduction of synchronization constraints to the language, the programmer will be better able to specify which methods may execute before completion of the continuation.

When the explicit continuation is executed, the variable specified as *ret-val* of type *ret-type* is given the return value of the invoked method. The type of this return value is type-checked dynamically, by Broadway, when the continuation is invoked.

Return The **return** statement is used by any method which was invoked with an explicit continuation method invocation. The **return** triggers the explicit continuation. Note that the **return** does not cause the method to exit. A value may be supplied to **return** of any type; this value will be type-checked dynamically before invocation of the continuation.

`return [<`*expression*`>];`

A **return** statement which is executed from a method which was not invoked with an explicit continuation will have no harmful effect on the system.

Operators There are many binary and unary operators in Screed. Below is a list of them. Operators currently work only on *int* and *real* types.

+ Addition,

− Subtraction or arithmetic negation,

∗ Multiplication,

/ Division,

| Logical OR,

& Logical AND,

! Logical negation,

< Less-than,

<= Less-than or equal,

> Greater-than,

>= Greater-than or equal,

== Equality (works for all types).

New To create new actors, the **new** construct must be used. **new** takes an actor-class name and an optional location and returns the address of the new actor.

`new <`*class-name*`> [ @ <`*location*`> ]`

As with the operators above, the **new** statement returns a value. Therefore, it may be used as the right-hand side of assignment statements or as an actual parameter to an actor invocation. The location is an *int* value which specifies a particular Broadway node. If no such location is specified, Broadway will determine placement in some default manner.

B.3 Data Types

There are six primitive data types in Screed. There is currently only one operator for the construction of more complex types, though others may be added in later versions. The primitive types are:

int Integer type,

real Floating point numbers (double precision),

string Character strings,

actor Actor addresses,

method Holders of method names,

behavior Holders of class values,

msg Messages

As shown above, variable declarations are always proceeded by a **var** statement. The **var** statement may then be followed by an arbitrary number of declarations of the form:

<primitive type> <variable-name>

The one type constructor is the array constructor. In Screed, arrays of any size and dimension may be declared. The declaration of arrays in Screed is as in C:

<primitive-type> <var-name>`[`*<dim1>*`]` `[`*<dim2>*`]`...;

In this case the '[' and ']' are actual operators and do not indicate optional text. All dimensions must be integer expressions and the resulting array's always have their first element at location 0.

B.3.1 Constants

Constants in Screed may be declared at the global level only. Each constant *must* have a type and must be initialized upon declaration. The syntax for constant declarations is:

`const` *<primitive type> <var-name>* `=` *<expression>*;

Constants may then be used anywhere following their declaration.

B.4 Predefined Names

Several built-in methods are inherited by every actor in the system. These methods provide various types of system-level support for each actor. Currently, there are no safeguards in Screed to prevent misuse of these methods although individual actors may overwrite them, thereby disabling them. These methods are:

dispatcher(actor dis) Set the meta-level dispatcher for the actor.

mailq(actor mq) Set the meta-level mailq for the actor.

migrate(actor wh) Migrate the actor to the same node as *wh*.

exception(actor except) Notify actor of the exception *except*.

Note that any of the above methods may be overwritten in the actor class definitions if a different behavior is desired. This is especially appropriate for the **exception** method which currently notifies the user of the exception and then crashes the node.

There is one class-name in Screed which has a reserved meaning. This is the `start` class. A `start` class should be defined with at least one method: *init*. An instance of this class will be automatically created by Broadway and an *init* method will be sent to it. Therefore, the *init* method acts as a *main()* in a C program; it is the entry point of the actor application. In all other ways, the class `start` and its instances behave as standard actor classes.

B.5 Library

To assist the Screed programmer, there are several predefined classes and actors. Rather than use special built-in function calls, these components provide system-level functionality using an interface (message passing) which is consistent with the rest of the language.

B.5.1 Predefined Actors

To provide I/O in a consistent and meaningful way, a special actor — whose address may be referenced as **io** — is accessible at all scopes. The actor has a *print* method which takes a variable number of arguments, the first of which is a format string. The rules for using the *print* method are identical to the C libraries *printf()* function. Which terminal the I/O appears on depends on the configuration of the system. However, once the system is configured, I/O will refer to the appropriate actor to perform output.

To support failure detection, there exists on each node a failure detector actor which may be reference by the name **detect**. Currently, the failure detector will only detect complete node failures. To use the failure detector, an actor sends a **subscribe** message to the actor **detect** with the address of the entity to notify in case of a crash, and the node which the actor is interested in monitoring. A -1 for the node number mean the actor is interested in all crashes. A subscription which would have the detector notify the subscribing actor if any node failed would appear as:

```
detect.subscribe(self,-1);
```

Upon detection of a failure, the detector will send an **exception** message to all appropriately subscribed actors. As explained below **exception** is a method built in to every actor.

B.5.2 Predefined Classes

Currently in Screed, there are two special classes besides those mentioned above for **io** and **detect**. The `Exception` class is the root class for all exceptions. It provides the basic functionality described in Chapter 3 for exception objects.

For timing analysis the `Timer` class exists. Instances of the class can be created to act as a stop-watch. In the future its role will be expanded to generate periodic messages to actors. The `Timer` class has three methods: *reset*, *stop* and *read*. *reset* resets the timer and starts it. *stop* stops the timer, and *read* returns the result in a variable of type real. To receive the value from *read*, and explicit continuation must be used.

Bibliography

[1] M. Acceta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A New Kernel Foundation for UNIX Developement. In *USENIX 1986 Summer Conference Proceedings*, June 1986.

[2] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.

[3] G. Agha. Concurrent Object-Oriented Programming. *Communications of the ACM*, 33(9):125–141, September 1990.

[4] G. Agha, S. Frølund, R. Panwar, and D. Sturman. A Linguistic Framework for the Dynamic Composition of DependabilityProtocols. In C.E. Landwehr, B. Randell, and L. Simoncini, editors, *Dependable Computing for Critical Applications 3*, volume 8 of *Dependable Computing and Fault-Tolerant Systems*, pages 345–363. IFIP Transactions, Springer-Verlag, 1993.

[5] K. P. Birman and T. A. Joseph. Communication Support for Reliable Distributed Computing. In *Fault-tolerant Distributed Computing.* Springer-Verlag, 1987.

[6] Roy Campbell, Nayeem Islam, David Raila, and Peter Madany. Designing amd Implementing Choices: An Object-Oriented System in C++. *Communications of the ACM*, pages 117–126, September 1993.

[7] E. Cooper. Programming Language Support for Multicast Communication in Distributed Systems. In *Tenth International Conference on Distributed Computer Systems*, 1990.

[8] Antonio Corradi, Paola Mello, and Antonio Natali. Error Recovery Mechanisms for Remote Procedure Call-Based Systems. In *8th Annual International Phoenix Conference on Computers and Communicaton Conference Proceedings*, pages 502–507, Phoenix, Arizona, March 1989. IEEE Computer Society Press.

[9] F. Cristain. Understanding Fault-tolerant Distributed Systems. *Communications of the ACM*, 34(2):56–78, 1991.

[10] Quian Cui and John Gannon. Data-Oriented Exception Handling in Ada. *IEEE Transactions on Software Engineering*, 18:98–106, May 1992.

[11] Christophe Dony. Improving Exception Handling with Object-Oriented Programming. In *Proceedings of the 14th Annual International Computer Software and Applications Conference*, pages 36–42, Chicago, 1990. IEEE Computer Society, IEEE.

[12] Christophe Dony, Jan Purchase, and Russel Winder. Exception Handling in Object-Oriented Systems. *OOPS Messanger*, 3(2):17–29, April 1992.

[13] Jeffrey L. Eppinger, Lily B. Mummert, and Alfred Z. Spector, editors. *CAMELOT AND AVALON: A Distributed Transaction Facility.* Morgan Kaufmann Publishers, Inc., 1991.

[14] Jacques Ferber and Jean-Pierre Briot. Design of a Concurrent Language for Distributed Artificial Intelligence. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, volume 2, pages 755–762. Institute for New Generation Computer Technology, 1988.

[15] S. Frølund. Inheritance of Synchronization Constraints in Concurrent Object-Oriented Programming Languages. In O. Lehrmann Madsen, editor, *ECOOP'92 European Conference on Object-Oriented Programming*, pages 185–196. Springer-Verlag, June 1992. Lecture Notes in Computer Science 615.

[16] S. Frølund and G. Agha. A Language Framework for Multi-Object Coordination. In *Proceedings of ECOOP 1993.* Springer Verlag, July 1993. LNCS 627.

[17] John B. Goodenough. Exception Handling: Issues and a Proposed Notation. *Communications of the ACM*, 18(12):683–696, December 1975.

[18] Daniel T. Huang and Ronald A. Olsson. An Exception Handling Mechanism for SR. *Computer Languages*, 15(3):163–176, 1990.

[19] Yuuji Ichisugi and Akinori Yonezawa. Exception Handling and Real Time Features in an Object-Oriented Concurrent Language. In A. Yonezawa and T. Ito, editors, *Concurrency: Theory, Language, and Architecture*, pages 92–109. Springer-Verlag, Oxford, UK, September 1989. LNCS 491.

[20] Barbara Liskov. Distributed Programming in Argus. *Communications of the ACM*, 31(3):300–312, March 1988.

[21] Barbara Liskov and Alan Snyder. Exception Handling in Clu. *IEEE Transactions on Software Engineering*, 5(6):546–558, November 1979.

[22] P. Maes. Computational Reflection. Technical Report 87-2, Artificial Intelligence Laboratory, Vrije University, 1987.

[23] Carl Manning. ACORE: The Design of a Core Actor Language and its Compiler. Master's thesis, MIT, Artificial Intelligence Laboratory, August 1987.

[24] S. Mishra, L. L. Peterson, and R. D. Schlichting. Consul: A communication Substrate for Fault-Tolerant Distributed Programs. Technical report, University of Arizona, Tucson, 1991.

[25] M. H. Olsen, E. Oskiewicz, and J. P. Warne. A Model for Interface Groups. In *Tenth Symposium on Reliable Distributed Systems*, Pisa, Italy, 1991.

[26] Richard D. Schlichting, Falviu Christian, and Titus D. M. Purdin. A Linguistic Approach to Failure Handling in Distributed Systems. In A. Avižienis and J.C. Laprie, editors, *Dependable Computing for Critical Applications*, pages 387–409. IFIP, Springer-Verlag, 1991.

[27] Richard D. Schlichting and Titus D. M. Purdin. Failure Handling in Distributed Programming Languages. In *Proceedings: Fifth Symposium on Reliability in Distributed Software and Database Systems*, pages 59–66, Los Angeles, CA, January 1986. IEEE Computer Society Press.

[28] Santosh Shrivastava, Graeme Dixon, and Graham Parrington. An Overview of the Arjuna Distributed Programming System. *IEEE Software*, pages 66–73, January 1991.

[29] B. C. Smith. Reflection and semantics in a procedural language. Technical Report 272, Massachusetts Institute of Technology. Laboratory for Computer Science, 1982.

[30] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, second edition, 1991.

[31] C. Tomlinson and V. Singh. Inheritance and Synchronization with Enabled-Sets. In *OOPSLA Proceedings*, 1989.

[32] T. Watanabe and A. Yonezawa. A Actor-Based Metalevel Arhitecture for Group-Wide Reflection. In J. W. deBakker, W. P. deRoever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages*, pages 405–425. Springer-Verlag, 1990. LNCS 489.

[33] C. T. Wilkes and R. J. LeBlanc. Distributed Locking: A Mechanism for Constructing Highly Available Objects. In *Seventh Symposium on Reliable Distributed Systems*, Ohio State University, Columbus, Ohio, 1988.

[34] Y. Yokote, A. Mitsuzawa, N. Fujinami, and M. Tokoro. The Muse Object Architecture: A New Operating System Structuring Concept. Technical Report SCSL-TR-91-002, Sony Computer Science Laboratory Inc., Feburary 1991.

[35] A. Yonezawa, editor. *ABCL An Object-Oriented Concurrent System*, chapter Reflection in an Object-Oriented Concurrent Language, pages 45–70. MIT Press, Cambridge, Mass., 1990.