

# **Building Robust Distributed Applications With Reflective Transformations<sup>1</sup>**

*Anh Nguyen-Tuong and Andrew S. Grimshaw*

Technical Report CS-97-26

Department of Computer Science  
University of Virginia  
Charlottesville VA 22903

Email: {nguyen | grimshaw}@virgina.edu

## **Abstract**

Several projects are currently underway to build the nation's next generation computing infrastructure. These projects are sometimes called metasystems projects and seek to provide the illusion of a single, unified, virtual computing environment to end users. We expect metasystems to consist of millions of hosts and billions of objects, and on this scale, resource failures will be the norm and no longer the exception. One of the technical challenges that must be solved before such systems become usable is the widespread adoption of fault-tolerance techniques for both system level services and user applications.

As part of the Legion metasystem project, we have developed an architecture for incorporating fault-tolerance techniques into user applications. Our approach is based on reflective dynamic transformation techniques that manipulate the control and data flow characteristics of an application to achieve the desired fault-tolerance policies. In this paper, we present our Reflective Graph & Event model (RGE) computation model and illustrate sample reflective transformations for incorporating exception handling and replication techniques into applications.

**Keywords:** worldwide virtual computer, metacomputing, metasystems, fault-tolerance, reflection, reflective transformations

---

<sup>1</sup> This work is performed in the context of the Legion project. More information about Legion project is available on the Web at <http://legion.virginia.edu>

## 1 Introduction

The advent of fast networks and the wide availability of computing resources enable the realization of powerful virtual computers, or metasystems, that will harness resources on a national or global scale. These metasystems will consist of millions of hosts and billions of objects, and on this scale, resource failures will be common. One of the technological challenges that must be solved before such virtual machines can be used is the widespread adoption of fault-tolerance techniques for system level services and user applications.

While the need for fault-tolerance is clear, less clear is how to build distributed applications that are robust, i.e., resilient to faults, in a manner accessible to *mainstream* programmers – programmers with little or no knowledge of fault-tolerance techniques. Fault-tolerance protocols are regarded widely as complex and difficult to implement correctly, and when one factors in the complexity of writing distributed or parallel applications (without regards to any fault-tolerance), the magnitude of the problem becomes evident.

As part of the Legion metacomputing project [21], we have developed an architecture for incorporating fault-tolerance techniques into user applications. Our approach is based on dynamic transformation techniques that manipulate the control and data flow characteristics of an application to achieve the desired fault-tolerance policies. We advocate the use of reflection [28][31] as a design principle for carrying out these transformations, hence the name *reflective transformations*. A reflective system is by nature self-referential: it has a representation of itself that it can observe and manipulate, and its representation is causally connected to its behavior.

Structuring applications along reflective guidelines is the key to enabling our transformation techniques. Consider the following representative constructs extracted from replication [45] and log-based rollback-recovery techniques:

- on the receipt of a message, log onto stable storage
- before sending a message, append a sequence number
- when receiving a message, inspect sequence number. If sequence number is less than X, discard message
- on the receipt of a message, multicast to replicas, collect replies and vote on the result

Such constructs can be mapped easily onto a reflective architecture provided that the architecture exports or reflects the act of sending and receiving messages. While others have proposed using reflection as the basis for building robust applications [1][16][39], these works have focussed on specific fault-tolerance techniques such as replication [16] or providing persistence [39]. [1] is a notable exception and its architecture provides support for both exception handling and replication.

In this paper, we specify a novel reflective computational model, the Reflective Graph & Event Model (RGE), that differs from previous works in its scope and the information that it reflects to protocol writers. RGE is designed to support a wide range of fault-tolerance policies, such as replication techniques, exception handling policies, and backward error recovery techniques. As its name implies, RGE is itself the composition of two computational models, a graph-driven execution model and an event-driven execution model. Salient features of RGE include the concepts of graphs as first-class entities that regulate control and data-flow between objects, events as regulating the control and data-flow within objects, and the correspondence between the graph- and event-driven models.

We will measure the success of the RGE model against the following criteria:

- *Application diversity*. A good solution should be applicable to a wide variety of applications. Note that this criterion is related to the next two – a solution that meets the coverage and reusability criteria should enable the construction of a diverse set of applications.
- *Coverage*. There is a large body of fault-tolerance literature available. A good solution should be able to accommodate at least the common fault-tolerance techniques in use today. Additionally, it should also enable new fault-tolerance techniques.

- *Reusability.* A good solution should encourage and enable fault-tolerance protocol writers to encapsulate their code in reusable forms, e.g., libraries or objects.
- *Localized cost.* Fault-tolerance is not free – incorporating fault-tolerance techniques adds time and resource consumption costs. Applications should only pay for the amount of fault-tolerance that they require. In the extreme case, applications that do not require fault-tolerance should not be negatively impacted by those that do.

Note that the above criteria apply to any solutions that attempt to incorporate fault-tolerance into user applications.

We present related work in section 2, followed by the system model in section 3. We present our reflective computation model (RGE) in section 4. In section 5 we present a novel exception propagation model based on RGE. In section 6 we express several replications techniques in terms of the RGE model. We conclude and present future works in section 7.

## 2 Related Work

To date no metasytems exist on a large scale except perhaps for the World Wide Web. Several projects are under way to build a metasytem, including Globus [18], Globe [50] and Legion [21]. While all recognize the need for fault-tolerance, specific details are sketchy or non-existent. The RGE model addresses the issue of incorporating fault-tolerance into applications and suggests a way of structuring robust distributed applications. Note that in many instances, system-level services provided by metasytems are themselves applications. Thus our work is not only relevant for user-level applications but also the construction of metasytems themselves.

Works on graph-based models (2.1), event-based models (2.2), reflective systems (2.4), and, in general, other approaches (2.3, 2.5, 2.6, 2.7) to support the construction of robust distributed applications, are all relevant to our own.

### 2.1 Data-Flow

RGE uses the Macro-Data Flow (MDF) model [23] as the basis for specifying control-flow and data-dependence information between objects. MDF is the basis for Mentat [22], a high-performance object-oriented system, as well as its successor, Legion, a worldwide metasytem software project. The functional nature of the data-flow model lends itself well to fault-tolerance techniques [2][26][37]. Other coarse-grained data-flow systems include Paralex [2], CDF [3], HeNCE [6], Code/Rope [12]. Our choice of MDF as a starting point for the RGE model stems from the fact that MDF is a proven model and further, it has a notion of persistent actors – actors that maintain state information between method invocations. While in theory any computations that can be modeled with persistent actors can also be modeled with pure functional actors (intuitively, any state information can be encapsulated by tokens), practical considerations lead us to a persistent actor abstraction.

However, in order to build components that transform the underlying computation to achieve robustness, it is not sufficient to just model computations with MDF or any other graph-based execution models. We argue that these models should possess reflective capabilities similar to those found in our RGE model.

### 2.2 Events

The event paradigm is well established and many systems use it as the basis for extensibility, i.e. Coyote [9], the Java Bean Component Model [24], SPIN [40], Legion [53]. While the RGE model does not mandate that objects be implemented using events, a natural implementation of RGE would use a system that is event-driven such as the protocol stack of objects in Legion.

We use the event abstraction within the RGE model to capture and reflect the “internals” of objects to fault-tolerance protocol writers. Events allow protocol writers to intercept and reroute both messages and method invocations. More importantly, associating events with the acts of receiving/sending

messages/methods allows protocol writers to express fault-tolerance algorithms in a natural way by treating messages as abstract entities. Furthermore, the RGE model specifies a correspondence between the internal implementation (events) of an object and its external data and control-flow (graphs).

## 2.3 Group communication

Multicast group communication primitives have been proposed as a useful foundation on which to build robust systems. Examples of such systems include Amoeba [27], Coyote [9], Delta-4 [41], Isis [10], Horus [43], and Totem [35]. These systems are also known as CATOCS (causally and totally ordered communication systems) and provide varying degrees of ordering properties in the presence of failures. This class of systems has been the source of much controversy and has been criticized on the basis of the end-to-end argument [13]. The basic argument against CATOCS revolves around the tradeoffs and mismatches between the needs of user applications and the costs of guaranteeing ordering property – applications that do not require CATOCS should not have to pay for it. Recent CATOCS systems such as Horus or Coyote partially answer this criticism – they provide an extensible and configurable protocol stack in which the ordering properties of the communication systems can be composed and/or modified in a modular fashion to suit the needs of the applications. For an overview of CATOCS systems, please see [42].

Our work with the RGE model is orthogonal to the issue of using group communication primitives. We focus on reflective transformations that manipulate the control and data-flow of objects and applications to obtain a desired fault-tolerance policy, and hence could easily incorporate group communication primitives as part of our transformations.

## 2.4 Reflection

Reflection has been used successfully in several contexts, including operating systems [8], programming languages [16][28], and in the construction of dependable systems [1][16][29][39]. Its use has also been advocated for the next generation of real-time global databases [46].

In [29][39], the Common List Object System (CLOS) [28] is extended to support persistence using reflection. In [16], reflective features of the language open-C++ are used to implement replication techniques for objects. MAUD [1] is a meta-level architecture for building adaptively dependable systems that has been implemented on an actor-based system. While these systems demonstrate the feasibility of using reflection as an enabling technology for structuring robust applications, the question as to what information should be reflected is an open one. We argue that at the very least, the control flow and data-dependencies of an application should be visible to protocol writers. The RGE model captures just that information and exports it to protocol writers through first-class program graphs and events.

Many distributed systems are already in some degrees reflective. CORBA's Dynamic Invocation Interface [7], Microsoft's Distributed Component Object Model (DCOM) [11], Java's Core Reflection API [34] all provide dynamic access to the methods and signatures exported by objects. However, these systems are best described as introspective as their support for reflection is limited, for example, they do not allow the method dispatch mechanism itself to be modified.

## 2.5 Middleware fault-tolerance

Middleware approaches to fault-tolerance can provide both flexibility to protocol designers and transparency to end users. As the name suggests, middleware solutions sit between the user program and the underlying system. The common trait with middleware solutions is that they intercept the control flow between the application level and the underlying system to implement the desired fault-tolerance guarantees. For example, Fault-Tolerant Mach [44] uses the concept of sentries to intercept calls between the application layer and the Mach system services. In [4], the author proposes a fault-tolerance layer that exports the same interface as the underlying operating system. In [36], the authors exploit the Internet Inter-Orb Protocol Interface (IIOP) to extend CORBA with fault-tolerance based on replication techniques. In [48], the authors specify a replication service for CORBA using a primary/backup scheme. Electra [32][33]

and Orbix+ISIS [25] are both CORBA-compliant systems that provide fault-tolerance using groups and group communication primitives (section 2.3).

Implicit in all these approaches is the definition of a sufficient set of information that needs to be intercepted, i.e. available, in order to implement a set of fault-tolerance policies. Viewed in this manner, the RGE model could be described as the specification of a middleware architecture for building robust applications.

## 2.6 Fault-tolerance encapsulated in objects

An approach to providing fault-tolerance at the user level is to use the object-oriented paradigm to encapsulate fault-tolerance properties. Arjuna [49], Avalon/C++ [15], DOME [5], BAST [19] are examples of systems that take this approach. In [16], the authors note that using inheritance alone poses problem for integrating replication techniques into user applications as standard object-oriented programming languages do not allow protocol writers to modify the method invocation dispatch mechanism. In [1], the authors observe that traditional metaobject protocols such as the one in [16] do not have explicit representations for the control flow of objects. The RGE model directly addresses both issues.

DOME provides a set of predefined classes to support data-parallel computations. By exploiting semantic information and the common structure of data-parallel computations, DOME provides support for object migration, load-balancing, and fault-tolerance through a heterogeneous checkpoint/restart mechanism. DOME does not attempt to satisfy the needs of all applications and it can thus exploit domain knowledge to provide an optimized set of services.

The issue of whether to use an object-oriented language to expose fault-tolerance properties is orthogonal to the RGE model. We do not expect the RGE model to be directly visible to end users. System builders such as language designers and compiler writers can use the RGE model as their implementation model and then export their own API to end users.

## 2.7 Exception handling

Exception handling in sequential programming languages is a well-understood area [17][20]. Examples of such languages include Ada [14], C++ [51], CLU [30]. While we draw inspiration from these works, we differ from these in that our exception propagation model is not language-specific. We make a distinction between exception handling (as espoused by a specific programming language) versus exception propagation.

In [1], the authors distinguish between two- and three-party systems with respect to exception handling. In a two-party system, exception handling routines are defined within the scope of invoking objects (invoker and signaler) whereas in a three-party system, the handler may be independent of the call chain. The authors note the inadequacies of two-party exception handling systems in obtaining a “global” view of the system – in a three-party system, monitoring of exceptions is easily modeled by an independent handler. Associating exceptions with their handlers is performed at the language level – users may attach handlers at the method, class or global level of granularities.

Using first-class program graphs to specify the propagation of exceptions, our model subsumes both the two- and three-party models (Section 5). Further, we do not specify a language for associating exceptions with handlers and instead choose to leave this task to language designers. Our approach is similar in respect to CORBA’s. In CORBA, objects can declare exceptions using the language neutral CORBA IDL. It is then the responsibility of a specific programming environment to map CORBA’s exceptions back to a specific programming language. Note that CORBA’s exception propagation semantics are limited – exceptions can only propagate back to the invoker of a method.

# 3 System Model

We base our work on the object-oriented paradigm. Objects have a system-wide name, have their own thread of control and address space, and communicate solely by issuing asynchronous method invocations.

While at the abstract level we reason in terms of communicating objects, we use the standard message-passing paradigm as the implementation model. While there may be other implementation models for realizing the object-oriented abstraction, e.g., shared memory, we feel that the message passing model is the most natural for a wide-area environment in which compute resources are geographically dispersed and linked by networks. The fact that we use message-passing as the implementation model for our object-oriented abstraction is not just an implementation detail, it is a primary feature of our approach. Recall that reflective systems are by definition introspective – in our case, the object-oriented abstraction can “look” inward and affect its own message-based implementation.

A distributed application consists of a set of interacting objects. A *robust* distributed application is one that can tolerate a given class of faults. We do not assume a specific fault-model – each transformation implicitly contains a set of assumptions regarding possible failure modes.

## 4 Reflective Graph and Event Model (RGE)

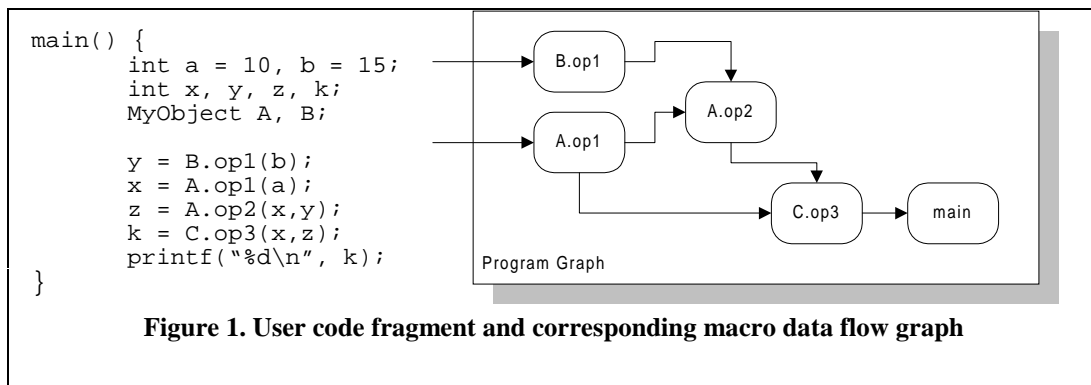
As the name indicates, the Reflective Graph and Event Model is composed of two other execution models: (1) a graph-driven execution model, macro-data flow (MDF), which is an extension of the data-flow model [52], and (2), a standard event-driven execution model. MDF specifies the interaction between objects while the event model specifies the internal implementation of objects. We present these models independently (Sections 4.1 and 4.2), and then jointly (Section 4.3), to simplify our exposition.

### 4.1 Reflective Graphs: Macro Data Flow

Macro Data Flow (MDF), an extended data flow model, has been applied successfully in both the high-performance parallel computing and wide-area distributed systems arena [3][6][12][21][23]. We select MDF as a starting point for our infrastructure since it presents several advantages for building reflective fault-tolerance components:

- MDF graphs specify the control flow and data dependencies between object invocations
- MDF graphs encapsulate information about the future calling sequence of an object. Such information is not generally available to fault-tolerance protocol writers. We expect to produce specialized algorithms that take advantage of this information.
- MDF graphs are first-class entities. They may be manipulated and transformed and hence enable the construction of generic graph transformers.

We provide a very brief review of MDF and of our extensions. Interested readers may refer to the literature for a thorough description [23].



**Figure 1. User code fragment and corresponding macro data flow graph**

Program graphs model computations in MDF. Graph nodes are called actors and represent method invocations on objects<sup>2</sup>; arcs denote data-dependencies between actors; and tokens flowing arcs represent data or control information. The rules for data flow are simple, when an actor has a token on each of its

<sup>2</sup> For pedagogical reasons we will assume that methods are single-valued, i.e., only one value is returned by a method even though multiple copies of the return values may be made. Extension to multi-valued methods is simple.

input arcs, it may “fire” or execute its method, and deposit a token on each of its output arcs. Actors in standard data flow are idempotent; presented with the same input tokens, an actor will always compute the same output tokens. This property makes data flow particularly well-suited for modeling pure functions. However, it is not practical for modeling functions that retain state information between function invocation. MDF extends data flow and allows for persistent actors – actors that can retain state between invocations.

Figure 1 illustrates a fragment of code and its corresponding graph representation. An interesting aspect of MDF is that each actor receives a copy of the entire program graph<sup>3</sup> when executing a method. The actor uses the graph to determine where to send its output tokens, i.e., return values. For example, when `A.op1` finishes executing, it inspects its copy of the program graph and forwards its return value on all outgoing arcs, i.e. towards, `A.op2` and `C.op3`.

#### 4.1.1 MDF Extensions

We extend the MDF model to incorporate the notions of errors and graph annotations.

**The  $\perp$  Token<sup>4</sup>.** Modeling exceptions is difficult in the current MDF model. To alleviate this problem, we introduce a special token,  $\perp$ , to denote an error value. The firing rules for actors remain the same: when a token is present on all its input arcs, the actor may fire. The difference is that the output tokens may either be regular tokens or the  $\perp$  token. In the presence of an error, the  $\perp$  token could thus propagate throughout a graph. For example, in Figure 1, if the actor `A.op2` cannot compute its return value, it may generate a  $\perp$  token on its output arc. The  $\perp$  token could then propagate all the way to the consumer of the computation, `main`. Alternatively, `C.op3` could mask the error and return a regular token to `main`.

**Graph annotations.** Another extension to MDF is the addition of graph annotations. Annotations may be attached to graphs, arcs or the nodes themselves. Annotations are 2-tuples, a variable name and its associated value, and provide a generic mechanism for capturing and accessing information dynamically. Example semantic information useful for fault-tolerance would be to know whether a given method is deterministic or non-deterministic, whether a given method updates state information, or whether a given object supports fault-tolerance protocol X.

To illustrate a possible use of graph annotations consider the implementation of a transformation that relies on the deterministic behavior of objects. Before carrying out a transformation, protocol writers can query the graph to determine if in fact this assumption holds. If the answer is negative or unknown, protocol writers can adapt and use an alternative transformation that does not rely on determinism. Alternatively, one could raise a system exception when the assumptions of the transformations are violated.

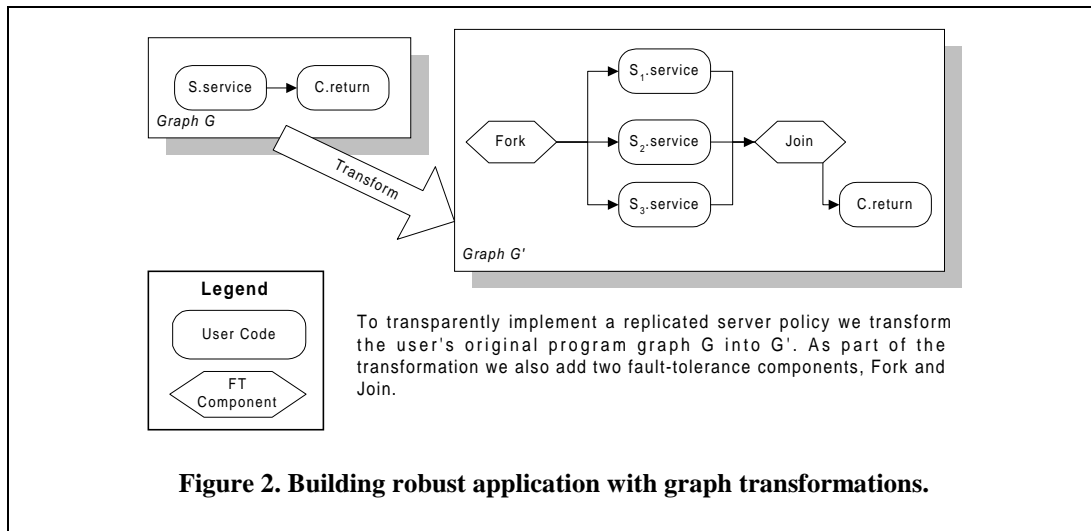
#### 4.1.2 Example

The following example demonstrates a representative reflective transformation. In Figure 2, the graph `G` represents a standard remote method invocation from a client object to a server object. The client issues a method invocation (`S.service`), and the server replies by invoking the return method on the client (`C.return`). Through a graph transformation (`G` to `G'`), we transparently replicate the method invocation such that the request is first sent to a `Fork` object, which in turn routes the request to the server replicas. The `Join` object collects the results from the server replicas and sends a single reply to the client. Note that the transformation makes use of two generic objects, `Fork` and `Join`.

---

<sup>3</sup> For this presentation we assume that the entire program graph is carried with each method invocation. In practice the size of the graph that needs to be carried on each invocation is optimized heavily.

<sup>4</sup> Pronounced “bottom” token.



## 4.2 Reflective Events

The event paradigm is well-understood and has been applied successfully in such diverse areas as windowing systems [38], extensible kernel systems [40], component based systems [24], and the building of flexible protocol stacks [9][53]. Here, we advocate the use of the event paradigm for building reflective fault-tolerance components for distributed systems.

The versatility of the event paradigm resides in its ability to decouple communication between various components of a system both temporally and spatially – features that are essential to component-based systems. Events provide a uniform infrastructure to bind components together. When component  $X$  wishes to announce to the system that something of interest has happened, it announces an event  $E$ . Components that have registered their event handlers with the event manager previously are notified of the event  $E$ . The handlers are then called immediately upon the announcement of  $E$  (synchronous), or alternatively, the execution of the handlers may be deferred (asynchronous). In addition, events may carry arbitrary data.

Before we describe the utility of events for building fault-tolerance components, we review their use in the context of building extensible protocol stack for objects in distributed environments. We will use the Legion system as an example. In Legion, objects are configurable along many dimensions including the programming language used to implement the object, its security policy, its performance characteristics, and its accounting policy. Reflecting this need for configurability and extensibility, the protocol stack for Legion objects is build by connecting components via the event paradigm. For a more detailed description of the Legion Event Model, see [53].

The most striking feature of the Legion Event Model is that only a few events are needed to build the protocol stack. They are:

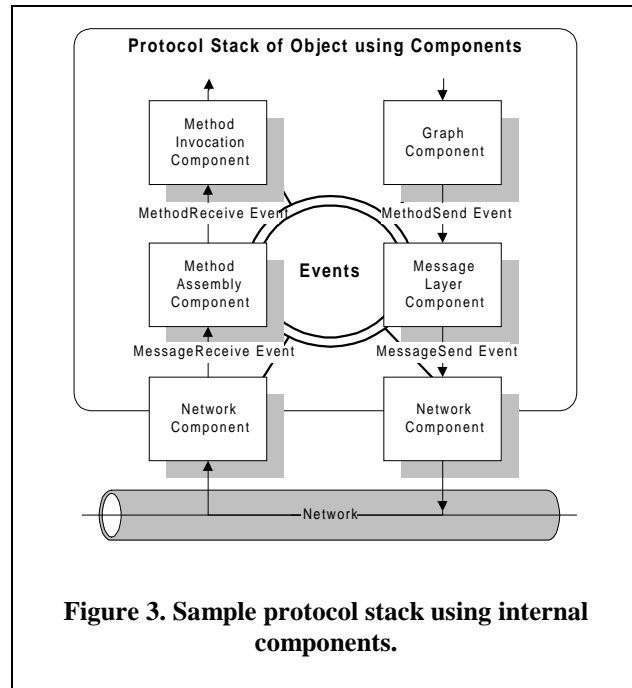
- **MessageReceive.** The object has received a fully formed message, i.e., all packets have been assembled
- **MessageSend.** The object is sending a fully formed message
- **MessageComplete.** The recipient of the message has acknowledged its receipt
- **MessageError.** The sending object was unable to send the message
- **MethodReceive.** The object has assembled a complete method invocation
- **MethodSend.** The object is invoking a method on another object
- **MethodComplete.** The called object has received the method invocation
- **MethodError.** The calling object is unable to invoke a method



These events form the core vocabulary set with which components may communicate with one another. This set is not specific to Legion; any object-oriented system implemented over a message passing environment, would have a similar set of building blocks, whether expressed in terms of events or otherwise<sup>5</sup>. The relatively small numbers of events should not detract from the expressive power of the event model; these events form a sufficient set to perform the following functions: encryption, decryption, authentication, signing, accounting, debugging, support for various programming languages, and method assembly. Figure 3 illustrates the protocol stack of the Legion run-time library configured using components.

The set of events previously identified is central to many fault-tolerance techniques. Many techniques can be viewed as wrappers around the underlying computation in the sense that they treat the computation as a black box with well-defined entry and exit points (Network Component in Figure 3). The assumption is that messages received via the entry points drive the computation, i.e., state transitions occur as the result of the receipt of messages. Similarly, sent messages cause other processes to make state transitions.

Events such as method receive and method send “reflect” the underlying structure of a computation and enable the writing of components that use this reflective capability to observe, intercept, and transform underlying computations.



#### 4.2.1 Example

The next example illustrates a representative event-based reflective transformation. In Figure 4, the network component announces a `MessageReceive` event and the handlers  $h_1$  through  $h_n$  are invoked. By inserting the handler  $h_0$  we transform the original control flow to log incoming messages onto stable storage before allowing the other handlers to be invoked.

### 4.3 Combining Graphs & Events

While the graph and event models have been presented separately, the RGE model specifies a correspondence between the external control and data flow of an object (with graphs) and its internal control and data flow (with events). Both aspects of the model are important as reflective transformations will need to take place within objects as well as externally to objects.

As an example of an event-based reflective transformation that requires access to program graphs, consider the implementation of the following majority voting algorithm: (1) intercept method invocations, (2) route the invocations to replica objects, (3) retrieve the results, (4) vote on the output, and (5) return the value to the caller.

<sup>5</sup> Note that the events presented fall within two broad categories: those that are related to the object-oriented abstractions and those that are related to the message-passing implementation model.

Within the RGE model, step 1 can be expressed as an event handler that manipulates program graphs (steps 2-5).

Conversely, protocol writers may wish to associate events with graphs, such that when an object announces an event, it automatically executes the corresponding program graph. The implication of associating graphs and events in this manner is that the behavior of an object now may be controlled externally – policies that often are embedded inside objects may now be set dynamically from the outside<sup>6</sup>.

In the next section, we illustrate an exception propagation model that fully exploits both aspects of the RGE model.

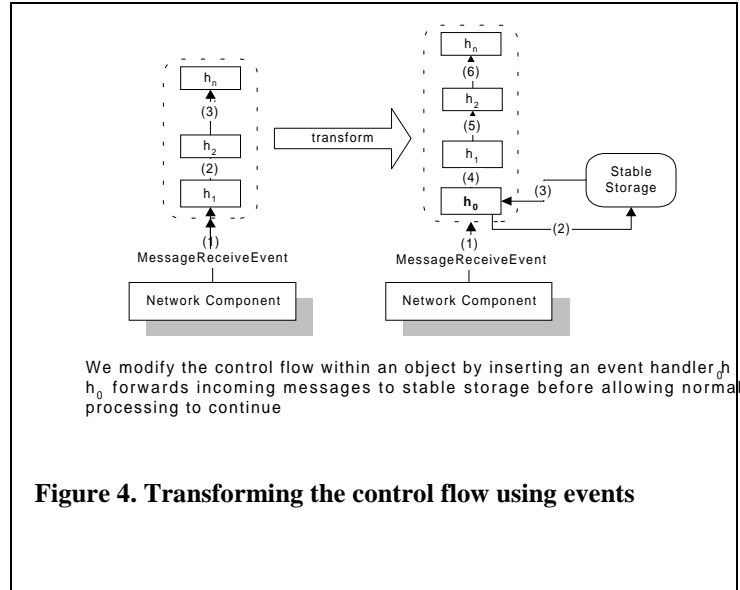


Figure 4. Transforming the control flow using events

## 5 Exception Propagation Model

Exceptions have been proposed as a structuring mechanism for building distributed robust applications [17] as they provide a mechanism for error detection and recovery. We now present a novel exception *propagation* model based on the RGE model. We distinguish between exception propagation and exception handling. We view exception handling as specific to a programming language or system, whereas exception propagation only deals with the propagation of exceptions between objects, regardless of the programming language used.

Common definitions for exception are: “errors or unusual conditions that are software-detectable or “any unusual event, erroneous or not, that is detectable either by hardware or software, and that may require special processing” [47]. The word exception has the connotation of a rare event, an assumption that no longer holds true in a wide-area distributed systems<sup>7</sup>. Examples of possible exceptions include:

- **Security violations:** the invoker of an object does not have proper authorization
- **Communication errors:** object X is unable to communicate with object Y
- **Binding errors:** object X is unable to find the network address for object Y
- **IDL errors:** a method invocation on object X does not match X’s exported interface
- **Resource errors:** object X does not have sufficient resources to service a request
- **Traditional errors:** floating point exceptions, divide by zero, out-of-range data, etc...
- **User-defined errors:** errors specific to the user code

In the following sections we present our exception propagation model and demonstrate its expressive power by mapping three different policies: flow-backward (5.2), flow-forward (5.3), and generic propagation (5.4). We then briefly describe an event-based mechanism for raising exceptions (5.5).

### 5.1 Model

We first define the following terms: exception, exception interest (EI) and exception interest set (EIS).

<sup>6</sup> Note that this feature may be undesirable in some instances. For example, a security-conscious object may not wish to have any of its policies set by an external source.

<sup>7</sup> We will continue to use the word exception within this report, however the term ‘notification’ would be more suitable.

- **Exception.** An Exception is a data structure that consists of a 2-tuple,  $\langle \text{ExceptionType}, \text{exceptionData} \rangle$ . The ExceptionType is composed of two values, a majorTypeID and a minorTypeID. The majorTypeID classifies exceptions within broad categories, e.g. security, communication, math. The minorTypeID further subdivides each category into subcategories, e.g. security:authentication, security:encryption, communication:network, communication:binding. The exceptionData field carries arbitrary data.
- **Exception Interest.** An exception interest is a 2-tuple,  $\langle \text{exceptionType}, \text{exceptionGraph} \rangle$ . The exception type specifies the kind of exceptions that one is interested in catching. The exceptionGraph specifies the computation to be executed if a match is made between a raised exception and an EI.
- **Exception Interest Set.** An exception interest set contains a set of exception interests.

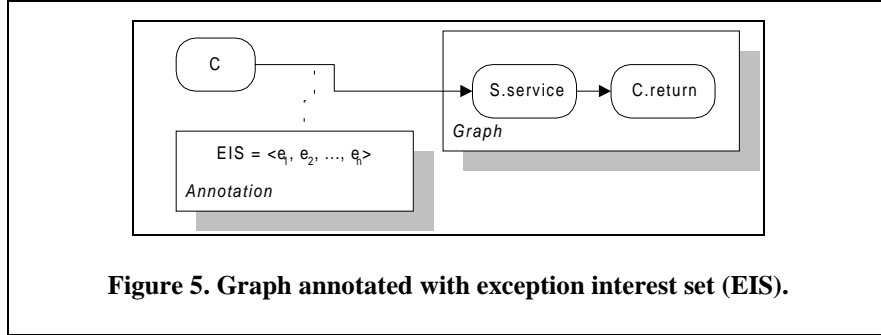


Figure 5. Graph annotated with exception interest set (EIS).

If an object wishes to register its interest in catching exceptions raised by objects in its future call chain<sup>8</sup>, it annotates its program graph by adding its exception interest in its EIS. When an object in the call chain raises an exception, it can inspect its EIS to look for a match. If there is a match between the exception raised by the object and an EI, the corresponding graph is then executed. If there are multiple matches, the order of graph execution is undefined.

We present mappings from the exception propagation model to the following propagation policies, flow-backward (Section 5.2), flow-forward (Section 5.3), generic propagation (Section 5.4). Within the flow-backward policy, we illustrate examples of exception masking (Table 2) and selective exception masking (Table 3).

## 5.2 Flow-Backward Policy

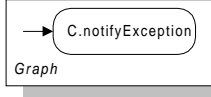
In a flow-backward propagation policy, the system uses the dynamic call chain of an application to propagate exceptions, e.g., a standard remote method invocation issued by a client on a server. When a server raises an exception, the system propagates the exception back to the client.

Consider the following code fragment and its program graph (Figure 5):

```
@Client: x = S.service();
```

If the client wishes to be notified of exceptions raised by `S.service`, it annotates the graph with the following exception interest:

<sup>8</sup> The future call chain of an object consists of all other objects that it calls directly or indirectly.

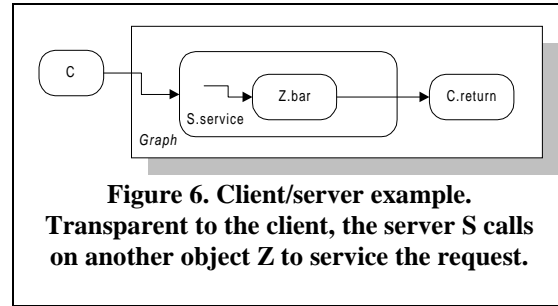
ExceptionType	Any	Any
ExceptionGraph		

**Table 1. Exception propagates back to originator of computation<sup>9</sup>**


The client sets the ExceptionType to Any. If S.service raises an exception, the method C.notifyException will be invoked.

Now consider a more complicated example where S.service itself calls on another object Z to perform its services .

```
@Client: x = S.service();
@S.service:y = Z.bar();
return y;
```



What should the annotation for Z.bar be? The answer depends on the desired policy. If the policy is to propagate exceptions back to the client then the annotation for Z.bar should be the same as that for S.service. On the other hand if S.service wishes to mask the exception from Z.bar, it annotates the graph for Z.bar with the following exception interest:

ExceptionType	Any	Any
ExceptionGraph		

**Table 2. Exception masking**

When Z.bar raises an exception it is propagated back to S. If S can take corrective actions then C need not be notified or even aware that there were ever any exceptions. On the other hand if S cannot handle the exception, then S can raise its own exception which will then propagate back to the client. This style of programming wherein computations are implemented with a series of client/server calls and wherein exceptions are masked in a hierarchical fashion is standard [17].

### 5.2.1 Selective masking of exceptions

In the examples thus far, objects have been interested in *all* exceptions. This need not be the case. For example, consider the example wherein a client is interested in math exceptions while the server object is interested only in security exceptions. To implement this policy the annotation at Z.bar contains the following EIS:

ExceptionType	Math	Any
---------------	------	-----

<sup>9</sup> Using notifyException as the function to invoke when an exception is raised is a convention. Each object may choose its own name or it may even have multiple functions for different types of exceptions.

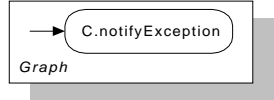

ExceptionGraph		
ExceptionType	Security	Any
ExceptionGraph		

Table 3. Selective exception masking

### 5.3 Flow-Forward Policy

In a flow-forward policy, exceptions propagate forward through the call chain instead of backwards. This policy only applies to computation models such as RGE that provide information about the future calling sequence of an application.

Consider the following example and its graph:

```
@A: x = D.op(B.bar(), C.foo());
```

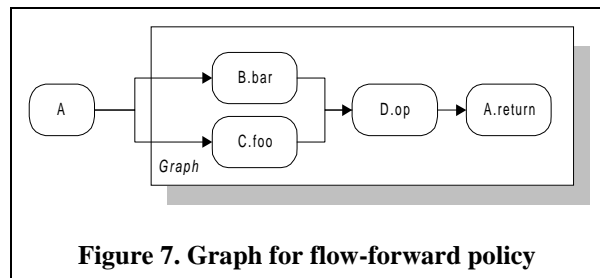


Figure 7. Graph for flow-forward policy

Assume that the method `B.bar` raises an exception. The exception propagates forward through the computation graph to `D.op` and then back to `A.return`. This example illustrates the utility of the flow-forward policy.

Recall that before `D.op` can execute it must first have a token on each of its input arc. If `D.op` receives only the token from `C.foo`, we say that `D.op` is a partial invocation. `D.op` must receive the token from `B.bar` in order to become a complete method invocation. Without the error token from `B.bar`, `D.op` would have to hold on to the token from `C.foo` indefinitely unless it uses a timeout scheme. By receiving the error token, `D.op` knows that an exception has occurred and can either mask the exception or immediately propagate the exception forward.

To realize this policy the object `A` annotates each arc in the graph with the following exception interest:

ExceptionType	Any	Any
ExceptionGraph	<Successor Node>	

Table 4. Exception interest for flow-forward propagation policy

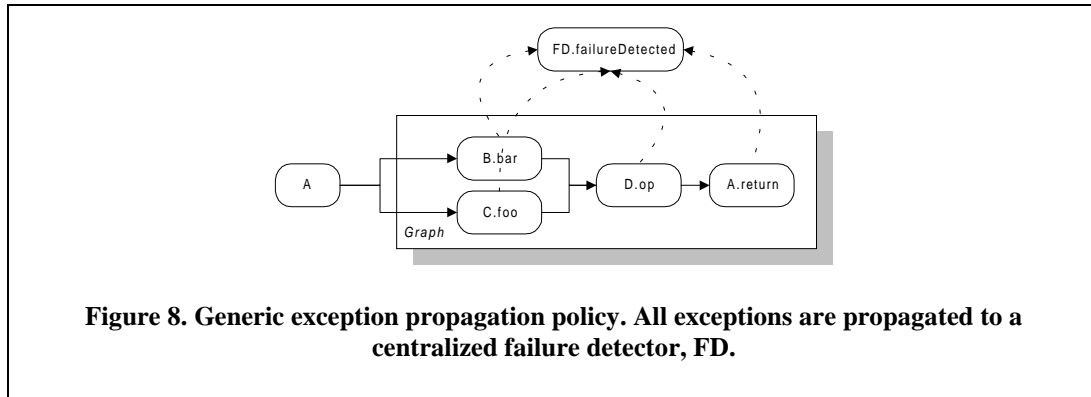
For example the input arcs to `B.bar` and `C.foo` would have as their successor nodes `D.op`. The input arcs to `D.op` would have `A.return` as the successor node.

### 5.4 Generic Propagation Policy

The previous two policies use the computation graph as the basis for propagating exceptions. The difference between the two was whether exceptions should propagate backward or forward through the graph. Another alternative is to exploit the expressive power of computation graphs and specify an arbitrary policy, one not necessarily tied to the computation graph.

Using the same computation graph as before, we want to propagate exceptions to a failure detector object, `FD`, that is not part of the computation. For example, if `B` is unable to communicate with `D` and raises a `CommunicationError` exception, we wish to notify `FD`. To implement this policy we annotate

the graph at each arc with an exception graph that corresponds to an invocation on method `failureDetected` of the FD object. Note that using a timeout exception as an object failure detection mechanism is applicable only if we assume a fail-stop failure model and no network partitioning.



## 5.5 Raising Exceptions

In all our examples, exceptions originate from within object, and thus naturally map onto the RGE model. We define a new event, the `ExceptionEvent`. To raise an exception, an object announces an `ExceptionEvent` and sets the data field to contain an `Exception` (`ExceptionType`, `exceptionData`). The handler for the `ExceptionEvent` performs a matching function – it inspects the current exception interest set and determines whether there is a match between the raised exception and any member of the EIS. If it finds a match, it extracts the exception graph and executes it. If it finds multiple matches, the order in which it executes the graphs is undefined. The EIS should be set when an object services a method invocation and unset when an object finishes servicing the invocation. This effect can be obtained if a system announces an event both on receipt and exit of a method invocation.

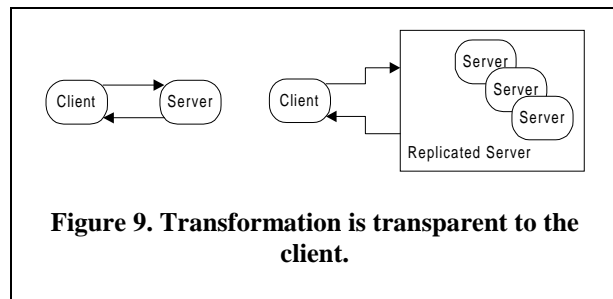
## 6 Replication

Replication is a common technique for achieving high availability, load-sharing, or high dependability. Using the RGE model, we now show sample reflective transformations for specifying replication strategies. Figure 9 illustrates an abstract view of our reflective transformations – in all cases the transformations are transparent to clients.

To simplify the presentation of our transformation techniques we assume:

- That the server is deterministic with respect to the sequence of method invocations<sup>10</sup>
- The existence of a name resolution service (NRS) that maps high-level names to physical network addresses

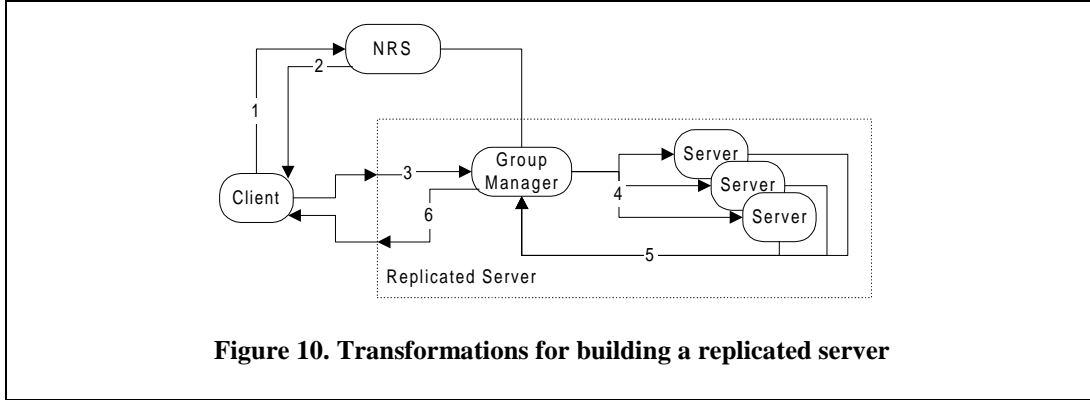
Figure 10 illustrates the general structure of our transformation. Before communicating with the server, the client must first obtain the server's physical address from the Name Resolution Service (NRS) (steps 1-2). Instead of returning the physical address for an actual server object, the NRS returns the physical name for a Group Manager object (step 3). The Group Manager is the key



<sup>10</sup> Recall that by using graph annotations (Section 4.1.1), the deterministic assumption can be checked prior to executing a transformation.

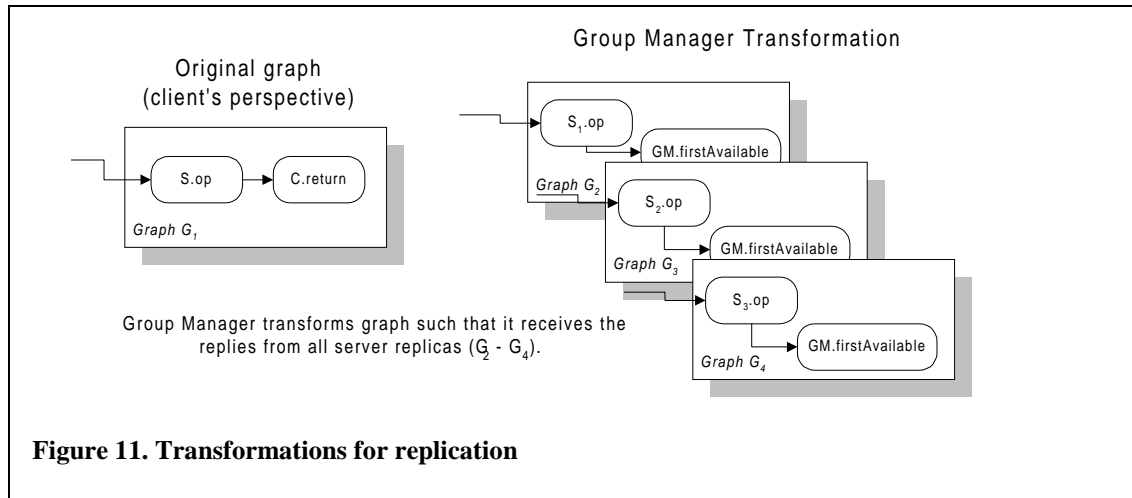
player in our transformation technique and is responsible for assembling method invocations on behalf of the `Server` and performing the necessary graph manipulations to implement various policies (steps 4-5). Finally, the client receives a reply to its request (steps 6).

To illustrate the advantages of having first-class program graphs, we show transformations that implement the following policies: active replication with fastest result returned (6.1), active replication with majority voting (6.2), load-sharing and replication for stateless objects (6.3). In all cases the transformations are transparent to the client.



### 6.1 Active replication with fastest result returned

In this policy method invocations from clients are sent to all replicas. The client receives the return value of the first replica to reply. An example usage of this policy would be to get good performance from servers located at different sites in the face of high variance in network bandwidth/latency.



In the general case the replicas should satisfy the *ordering* property – they should agree on the order of method invocations when servicing multiple clients. Furthermore, since clients may not be able to deal with multiple replies, only one should be sent.

Consider the following remote method invocation on a server object `S`:

```
@client: y = S.op();
```

Referring back to Figure 10, the NRS hands out the physical address of a group manager object, GM, instead of the actual server object. The GM is responsible for assembling the method invocation and implementing the replication policy.

Upon receiving a method invocation the GM saves the program graph ( $G_1$  in Figure 11) in an internal table, transforms the program graph ( $G_2 - G_4$  in Figure 11) such that return values from the replicas are sent back to itself instead of directly to the client, and multicasts the method invocation to all replicas. When the first return result arrives, the GM extracts the original program graph from its internal table to determine where to forward the result, and discards all subsequent replies.

## 6.2 Active replication with majority voting

Instead of returning the fastest reply to come back from the server replicas as in the previous policy we wait for all replies to arrive and perform a majority vote to determine the reply's correct value. The graph transformation needed to implement this policy is similar to the previous one. The GM performs the majority vote and returns the result to the client. Alternatively, instead of waiting for all values to return, the GM could wait until a majority agrees. If there is no majority, the GM raises an exception.

Another solution is to transform the original graph to use a separate `Voter` object. The `Voter` is then responsible for routing the correct return value to the client.

## 6.3 Load-sharing & fault-tolerance for stateless servers

Stateless servers represent an important class of servers as they lend themselves well to optimized replication techniques for both fault-tolerance and load-sharing [37]. Since by definition there is no server state to maintain, method invocations may be routed to any of the replicas without worrying about state consistency.

Similarly to the previous policies, the GM transparently intercepts `Client` requests. Once the GM has received a complete method invocation it uses a scheduling policy, e.g., round-robin or random, to select the replica to which it will forward the method invocation. There is no need to transform the program graph since the replicas will directly forward return values back to the client.

The previous approach does not deal with failure. If a stateless server replica fails while processing a method invocation, the client may never receive its expected reply. To solve this problem, we extend the algorithm to perform the following steps:

- GM stores method invocation from clients on stable storage.
- Server replicas notify GM that they are done servicing a method invocation. GM can then remove the invocation from stable storage.
- GM reissues lost computations.

# 7 Conclusion & Future Works

We have proposed the Reflective Graph & Event (RGE) model as the basis for a software infrastructure in which protocol designers can encapsulate a wide range of fault-tolerance techniques within reflective transformations. Salient features of the RGE model include the concept of first-class program graphs to specify the control and data flow between objects, and events to specify the control and data flow within objects. Using the RGE model, we have specified a novel exception propagation model that encompasses standard exception handling techniques and enables generic exception handling policies. Further, we have demonstrated reflective transformations to implement replication techniques. While it is too early to determine whether we have met our goals of *application diversity*, *coverage*, *reusability*, and *localized cost*, we believe that the RGE model provides a solid starting point.

The long-term goal of our research is to make building robust distributed applications within reach of mainstream programmers. To meet our goal, we have chosen an indirect approach. We have targeted our reflective architecture and the ensuing transformation techniques towards protocol writers and systems



designers. In turn, they will provide a suitable API for mainstream programmers. Note that our transformations are both generic and reusable. Over time, we expect to develop a set of fault-tolerance component libraries that may be reused by the community at large.

We are in the process of mapping several standard fault-tolerance techniques onto the RGE model and implementing them onto the Legion system. Specifically, we intend to provide support for backward-error recovery techniques (message logging, checkpointing), replication techniques (primary/backup, active replication), and techniques based on our exception propagation model. Together these techniques span the taxonomy of fault-tolerance algorithms. In addition, we believe that the RGE model will enable novel techniques that will take full advantage of its reflective capabilities.

## 8 References

- [1] G. Agha, D. C. Sturman, "A Methodology for Adapting Patterns of Faults", *Foundations of Dependable Computing: Models and Frameworks for Dependable Systems*, Kluwer Academic Publishers, Vol. 1, pp. 23-60, 1994.
- [2] O. Babaoglu et. al., "Paralex: An Environment for Parallel Programming in Distributed Systems", *Technical Report UBLCS-92-4*, Laboratory for Computer Science, University of Bologna, Oct. 1992.
- [3] R. F. Babb, "Parallel Processing with Large-Grain Data Flow Techniques", *IEEE Computer*, pp. 55-61, July 1984.
- [4] T. Becker, "Application-Transparent Fault Tolerance in Distributed Systems ", *Proceedings of the 2<sup>nd</sup> International Workshop on Configurable Distributed Systems*, pp. 36-45, 1994.
- [5] A. Beguelin, et. al., "Application Level Fault Tolerance in Heterogenous Networks of Workstations", *to appear in a special issue of the Journal of Parallel and Distributed Computing on Workstation Clusters and Network-based Computing*, September 1997.
- [6] A. Beguelin et. al., "HeNCE: Graphical Development Tools for Network-Based Concurrent Computing", *Proceedings SHPCC-92*, pp. 129-36, Williamsburg, VA, May 1992.
- [7] R. Ben-Natan, "CORBA: A Guide to the Common Object Request Broker Architecture ", *McGraw-Hill*, 1995.
- [8] B. Bershad et. al., "Extensibility, Safety and Performance in the SPIN Operating System", *Proceedings of the 15<sup>th</sup> ACM Symposium on Operating System Principles (SOSP-15)*, pp. 267-284, Copper Mountain, CO, 1995.
- [9] N. T. Bhatti, et. al., "Coyote: A System for Constructing Fine-Grain Configurable Communication Services", *Department of Computer Science Technical Report TR 97-12*, University of Arizona, July 1997.
- [10] K. P. Birman and R. V. Renesse, "Reliable Distributed Computing with the Isis Toolkit ", *IEEE Computer Society Press*, Los Alamitos, California, 1994.
- [11] N. Brown and C. Kindel, "Distributed Component Object Model Protocol – DCOM/1.0", <http://www.microsoft.com/oledev/olecom/>, Internet Draft, May 1996.
- [12] J. C. Browne, T. Lee and J. Werth, "Experimental Evaluation of a Reusability-Oriented Parallel Programming Environment", *IEEE Transactions on Software Engineering*, pp. 111-120, February 1990.
- [13] D. Cheriton and D. Skeen, "Understanding the Limitations of Causally and Totally Ordered Communication", *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles (SOSP-13)*, New York, ACM Press, pp. 44-57, December 1993.
- [14] Q. Cui and J. Gannon, "Data-Oriented Exception Handling in Ada ", *IEEE Transactions on Software Engineering*, pp. 98-106, May 1992.
- [15] D. L. Detlefs, M. P. Herlihy and J. M. Wing, "Inheritance of Synchronization and Recovery Properties in Avalon/C++", *IEEE Computer*, pp. 57-69, December 1988.
- [16] J. C. Fabre et. al., "Implementing Fault Tolerant Applications using Reflective Object-Oriented Programming", *The Twenty-fifth Symposium on Fault-Tolerant Computing (FTCS-25)*, pp. 489-498, 1995.
- [17] F. Christian, "Exception Handling and Tolerance of Software Faults", *Software Fault Tolerance*, M. Lyu Editor, Wiley, 1995, pp. 81-107.
- [18] I. Foster and C. Kesselman, "Globus: A metacomputing infrastructure toolkit", *International Journal of Supercomputing Applications*, 1997.

- [19] B. Garbinato, P. Felber and R. Guerraoui, "Modeling Protocols as Objects for Structuring Reliable Distributed Systems", *Proceedings of the Communications Networks and Distributed Systems Modeling and Simulation Conference (CNDIS'97)*, Phoenix, Arizona, January 1997.
- [20] J. B. Goodenough, "Exception Handling: Issues and a Proposed Notation", *Communications of the ACM*, 18:12, pp. 683-696, December 1975.
- [21] A. S. Grimshaw, "The Legion vision of a worldwide virtual computer", *Communications of the ACM*, 40:1, pp. 39-45, January 1997.
- [22] A. S. Grimshaw, A. Ferrari and E. West, "Mentat", *Parallel Programming Using C++*, The MIT Press, Cambridge, Massachusetts, pp. 383-427, 1996.
- [23] A. S. Grimshaw, J. B. Weissman and T. Strayer, "Portable Run-Time Support for Dynamic Object-Oriented Parallel Processing", *Technical Report CS-93-40*, Department of Computer Science, University of Virginia, July 14, 1993.
- [24] G. Hamilton (editor), "JavaBeans™", <http://splash.javasoft.com/beans/docs/spec.html>, Sun Microsystems, July 1997.
- [25] Iona Ltd. And Isis Distributed Systems, Inc., "An Introduction to Orbix+Isis", 1995. Available at [info@iona.ie](mailto:info@iona.ie).
- [26] R. Jagannathan and E. A. Ashcroft, "Fault Tolerance in Parallel Implementations of Functional Languages", *The Twenty-first Symposium on Fault-Tolerance Computing (FTCS-21)*, pp. 256-263, 1991.
- [27] M. F. Kaashoek and A. S. Tanenbaum, "Group communication in the Amoeba distributed operating system", *Proceedings of the 11<sup>th</sup> IEEE International Conference on Distributed Computing Systems*, pp. 222-230, May 1991.
- [28] G. Kiczales, J. D. Rivieres and D. G. Bobrow, "The Art of the Metaobject Protocol", MIT Press, 1991.
- [29] A. H. Lee and J. L. Zachary, "Reflections on metaprogramming", *IEEE Transactions on Software Engineering*, vol. 21, pp. 883-892, November 1995.
- [30] B. Liskov and A. Snyder, "Exception Handling in CLU", *IEEE Transactions on Software Engineering*, pp. 546-558, November 1979.
- [31] P. Maes, "Concepts and Experiments in Computational Reflection", *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pp. 147-55, October 1987.
- [32] S. Maffeis, "Adding Group Communication and Fault Tolerance to CORBA", *Proceedings of the 1995 USENIX Conference on Object-Oriented Technologies*, Monterey, CA, June 1995.
- [33] S. Maffeis, "A Fault-Tolerant CORBA Name Server", *Proceedings of the 15th International Symposium on Reliable and Distributed Systems (SRDS-15)*, pp. 188-197, October 1996.
- [34] Sun Microsystems, "Reflection", <http://www.javasoft.com/products/jdk/1.1/docs/guide/reflection/>, 1997.
- [35] L. E. Moser, et. al., "Totem: A Fault-Tolerant Multicast Group Communication System", *Communications of the ACM*, 39:4, pp. 54-63, April 1996.
- [36] P. Narasimhan, L. E. Moser, P. M. Melliar-Smith, "Exploiting the Internet Inter-ORB Protocol Interface to Provide CORBA with Fault Tolerance", *Proceedings of the Third USENIX Conference on Object-Oriented Technologies (COOTS)*, June 1997.
- [37] A. Nguyen-Tuong, A. S. Grimshaw, and M. Hyett, "Exploiting Data-Flow for Fault-Tolerance in a Wide-Area Parallel System", *Proceedings of the 15th International Symposium on Reliable and Distributed Systems (SRDS-15)*, pp. 1-11, October 1996.
- [38] A. Nye, "Xlib Programming Manual for Version 11", *O'Reilly & Associates, Inc.*, Volume 1, 1988.
- [39] A. Paepcke, "PCLOS: Stress testing CLOS: Experiencing the metaobject protocol", *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, 1990.
- [40] P. Pardyak and B. Bershad, "Dynamic Binding for an Extensible System", *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Seattle, WA, pp. 201-212, October 1996.
- [41] D. Powell, ed., "Delta-4: A Generic Architecture for Dependable Distributed Computing", *Springer-Verlag*, Berlin and New York, 1991.
- [42] D. Powell, "Introduction to Special Section on Group Communication", *Communications of the ACM*, 39:4, pp. 50-53, April 1996.

- [43] R. V. Renesse, K. P. Birman and S. Maffei, "Horus: A Flexible Group Communication System", *Communications of the ACM*, 39:4, pp. 76-83, April 1996.
- [44] M. Russinovich, Z. Segall, and D. Siewiorek, "Application Transparent Fault Management in Fault Tolerant Mach", *Foundations of Dependable Computing: System Implementation*, pp. 215-241.
- [45] F. Schneider, "Implementing Fault-Tolerant Services Using the State Machine Approach", *ACM Computing Surveys*, December, 1990.
- [46] J. A. Stankovic, S. H. Son, J. Liebeherr, "BeeHive: Global Multimedia Database Support for Dependable, Real-Time Applications", *Technical Report CS-97-08*, Department of Computer Science, University of Virginia, 1997.
- [47] R. W. Sebesta, "Concepts of Programming Languages", 2nd ed., *The Benjamin/Cummings Publishing Company, Inc*, 1993.
- [48] G. Sheu, et. al., "A Fault-Tolerant Object Service on CORBA", *Proceedings of the 17<sup>th</sup> International Conference on Distributed Computing Systems*, pp. 393-400, May 1997.
- [49] S. K. Shrivastava, G. N. Dixon and G. D. Parrington, "An Overview of the Arjuna Distributed Programming System", *IEEE Software*, pp. 66-73, January 1991.
- [50] M. van Steen, P. Homburg, and A.S. Tanenbaum. "The Architectural Design of Globe: A Wide-Area Distributed System", *Technical Report IR-422*, Department of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, March 1997.
- [51] B. Stroustrup, "The C++ Programming Language", Addison-Wesley, 2<sup>nd</sup> edition, 1991.
- [52] H. Veen, "Dataflow Machine Architecture," *ACM Computing Surveys*, pp. 365-396, vol. 18, no. 4, December, 1986.
- [53] C. L. Viles et. al., "Enabling Flexibility in the Legion Run-Time Library", *International Conference on Parallel and Distributed Processing Techniques (PDPTA '97)*, Las Vegas, NV, 1997.