# Using Reflection for Incorporating Fault-Tolerance Techniques into Distributed Applications*

Anh Nguyen-Tuong and Andrew S. Grimshaw

University of Virginia Department of Computer Science
{nguyen,grimshaw}@virginia.edu
http://legion.virginia.edu

## Abstract

As part of the Legion metacomputing project, we have developed a reflective model, the Reflective Graph & Event (RGE) model, for incorporating functionality into applications. In this paper we apply the RGE model to the problem of making applications more robust to failures. RGE encourages system developers to express fault-tolerance algorithms in terms of transformations on the data structures that represent computations—messages and methods—hence enabling the construction of generic and reusable fault-tolerance components. We illustrate the expressive power of RGE by encapsulating the following fault-tolerance techniques into RGE components: two-phase commit distributed checkpointing, passive replication, pessimistic method logging, and forward recovery.

## 1 Introduction

The advent of fast networks and the wide availability of computing resources make possible the realization of powerful virtual computers, or metasystems, that harness resources on a national or global scale. One of the technological challenges that must be solved before such virtual machines can be used in production mode is the adoption of fault-tolerance techniques for system-level services and user applications. Unfortunately, fault-tolerance protocols are widely regarded as complex. Implementing them *correctly* is likely to overwhelm all but the best programmers.

Our approach to remedying this problem is to view fault-tolerant applications as the sum of three parts: the application, the fault-tolerance technique, and the infrastructure required to enable their composition. Application programmers should focus on writing applications while fault-tolerance experts should encapsulate algorithms inside components. Within the context of the Legion metacomputing project, we have developed a reflective computational model, the Reflective Graph and Event (RGE) model, for enabling the composition of fault-tolerance techniques with user applications [11][13].

The basic design philosophy behind a reflective architecture is to expose—instead of hide—the elements that make up the structure of the system to developers. A reflective system is introspective; the system has a representation of itself that it can observe—its self-representation. Often, the self-representation of a reflective architecture is expressed in terms of abstract entities that may be manipulated to modify the behavior of the system. Thus, a reflective system promotes the writing of generic and reusable components that manipulate the self-representation. Such components may be written by domain experts and incorporated transparently into user applications. For example, Fabre *et al.* use a reflective programming language to incorporate fault-tolerance techniques into non-fault-tolerant applications [8], thereby freeing application programmers from the complex and error-prone task of implementing fault-tolerance algorithms.

In this paper, we demonstrate the applicability of the RGE model in encapsulating the following fault-tolerance techniques: distributed checkpointing, passive replication, pessimistic method logging, and forward recovery. The RGE model enables the manipulation of user computations at an abstract level by representing them as events, event handlers and program graphs [24]. These data structures are the self-representation of our reflective architecture and manipulating them is the basis for expressing fault-

*This work is partially supported by DARPA (Navy) contract # N66001-96-C-8527, DOE grant DE-FD02-96ER25290, DOE contract Sandia LD-9391, Northrup-Grumman (for the DoD HPCMOD/PET program), DOE D459000-16-3C and DARPA (GA) SC H607305A.

tolerance algorithms. The advantages of using an event-based architecture are well-known: components are decoupled from one another spatially and temporally, and they may be added/removed dynamically. Developers may extend object functionality by registering handlers with the appropriate events and by defining new events. A novel feature of the RGE event mechanism is that handlers may be executable program graphs that specify method invocations on remote objects. Graphs may be bound to their associated events at run-time, enabling the dynamic composition of functionality to objects.

The paper is organized as follows. We present related work in Section 2 and introduce the Legion system model in Section 3. We provide an overview of the RGE model in Section 4 and apply the model to encapsulate fault-tolerance techniques in Section 5. We conclude in Section 6.

## 2 Related Work

The RGE model provides a blueprint for structuring distributed applications based on reflective principles. The concept of reflection is not novel; its use has been advocated in several contexts, including programming languages [19][22], soft real-time systems [16], real-time global databases [27], agent-based systems [6], and in general, to incorporate non-functional requirements into user applications [28].

Reflection has also been used to incorporate fault-tolerance techniques into applications. Lee extends the Common List Object System [19] to support persistence using reflection in [20]. Fabre exploits reflective features of the language open-C++ to incorporate replication techniques into applications transparently [8]. MAUD is a meta-level architecture for building adaptively dependable systems that has been implemented on an actor-based system [1]. To our knowledge, RGE is the only reflective model that uses graphs and events as data structures for representing computations.

The event paradigm is well established and many systems use it as the basis for extensibility, e.g., Coyote [5], the Java Bean Component Model [29], SPIN [26], and Ensemble [15]. We use the event abstraction within the RGE model to capture and reflect the "internals" of objects to programmers. Events allow programmers 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 many algorithms in a natural way by treating messages as abstract entities. Furthermore, RGE events may be associated with graph handlers dynamically—enabling the run-time binding of functionality to objects. Graphs used in RGE are the embodiment of the Macro-Data Flow model [14]. Other data-flow systems include Paralex [2], CDF [3], HeNCE [4], Mentat [12][23] and Code/Rope [7]. Of these, Paralex and Mentat support replication. Unlike most graph systems, RGE graphs are exposed to system developers; they can be assembled dynamically and executed remotely. Graphs are reflective: graphs are the self-representation of a computation and transforming them has a direct impact on the future of a computation.

Globus is another metasystem project [9]. The primary difference between Globus and Legion is a philosophical one: Globus employs a "sum-of-service" approach for supporting users and specifies standard interfaces for such functions as security and resource management. Legion employs an "architecture" approach—system developers target a unified model that enables component reuse and interoperability. To our knowledge Globus does not provide integrated support for incorporating fault-tolerance techniques into user applications. Instead, application writers may use a hearbeat monitoring service as a base for implementing fault-tolerance techniques. Note that the two approaches are not mutually exclusive—RGE fault-tolerance components can make use of an external failure detecting service.

## 3 System Model

Legion is based on an object model of computing. Legion objects encapsulate both hardware and software resources. Objects are logically independent collections of data and associated methods with disjoint address spaces. Objects can contain one or more associated threads of control, and communicate via asynchronous method invocations. Objects are named entities identified by a location-independent Legion Object IDentifer (LOID) and are mapped to Legion Object Addresses (LOA) for actual communication. The LOA of an object includes the necessary information to communicate with it for remote method invocation, e.g., the IP address and port number.

Objects are persistent and can be in one of two states: active or inert. Active objects contain one or more threads of control and are ready to service method calls. Inert objects exist on persistent storage as passive object state representations (OPR) organized in a directory structure. Legion moves objects

between active and inert states to use resources efficiently, to support object mobility, and to enable failure resilience.

Every Legion object is defined and managed by its *class object*. Class objects in Legion are themselves active objects, and are given system-level responsibility. They create new instances; schedule, activate, and deactivate their instances; and assist client objects in locating instances of the class.

For a detailed description of the Legion object model, please see Grimshaw [13].

## 4 Reflective Graph and Event Model

As the name indicates, RGE uses graphs and events to specify and represent user computations. We provide an overview of graphs in Section 4.1, events in Section 4.2, followed by a discussion of *exoevents*—events whose handlers are graphs—in Section 4.3. For a more detailed presentation of the RGE model, please see Nguyen-Tuong *et al.* [24].

### 4.1 Graphs

Our use of graphs originated in the Mentat project, a high-performance object-oriented parallel processing system [12]. Graphs are the embodiment of the Macro-Data Flow model, an extension of pure data flow designed for coarse-grained parallel processing. For more details on Macro-Data Flow and how it is used to exploit opportunities for parallelism please see Grimshaw *et al.* [14].

Graphs specify method invocations and data dependencies between objects. Graph nodes are called actors and represent method invocation on objects, arcs denote data-dependencies between actors, and tokens flowing across arcs represent data or control information. When an actor has a token on each of its input arcs, it may execute its corresponding method, and deposit a result token on each output arc. Figure 1 illustrates a fragment of code and the corresponding graph representation.

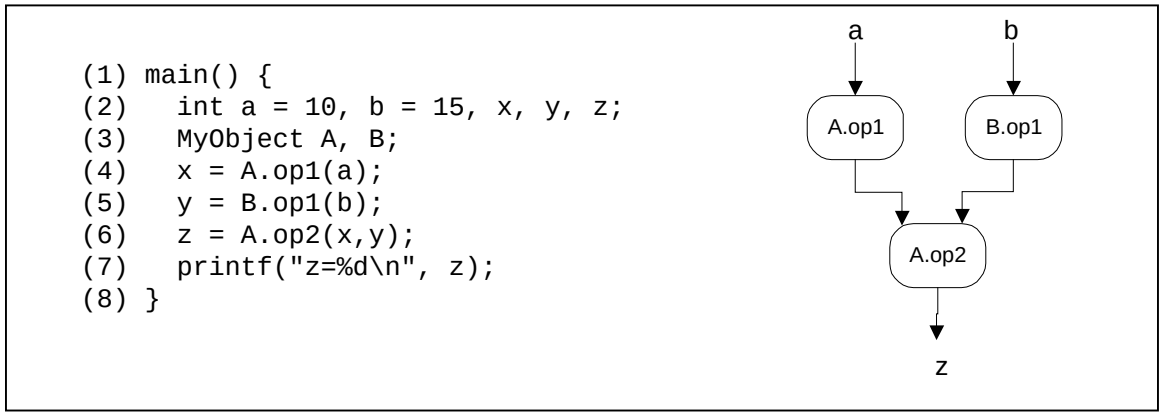


**Figure 1.** Sample code fragment and corresponding RGE program graph

Unlike a traditional client/server model, the results from the method invocations on lines 4 and 5 do not return to the `Main` object.[1] Instead they are forwarded directly to `A.op2`. When `A.op1` and `B.op1` execute, they each receive a logical copy of the graph.[2] The graph specifies where they should send their return values, namely, `A.op2`. Thus, graphs are reflective data structures that represent the future flow of the current computation. Graphs are first-class entities and may be assembled at run-time, transformed, passed as arguments to other objects, and executed remotely. Graphs enable system developers to build objects that adapt to their environment by assembling the proper method invocations dynamically and modifying the future flow of the computation.

Graphs may be annotated with <name, type, value> triples. The name field is simply a generic string, the type field indicates the type, and the value field consists of arbitrary data. The name and type fields dictate the interpretation of the value field. Annotations are properties tied to individual arcs and nodes,

[1] A client/server call is a special case of a 2-node graph: one for the server and the other for the return value.

[2] In practice, we only send the subset of the graph required for future computations, i.e., the transitive set of reachable graph nodes.

e.g., “Architecture=C90”, “Memory Usage=20MB”, “Semantic Property=Stateless”, and denote meta-level information. Annotations may propagate through the object method invocation chain, in which case we call them *implicit parameters*. If object A annotates its graph with an implicit parameter, invokes a method on object B, and B invokes a method on object C, A’s implicit parameter propagates to C. Implicit parameters provide a mechanism for adding meta-level information transitively and are similar to CORBA’s contexts [25]. The primary difference with CORBA’s contexts is that implicit parameters propagate automatically through the method invocation call chain.

### 4.2 Events

The event paradigm provides a well-understood mechanism for adding new functionality to objects. The versatility of the event paradigm resides in its ability to decouple communication between various components of a system both temporally and spatially—essential features of a 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.

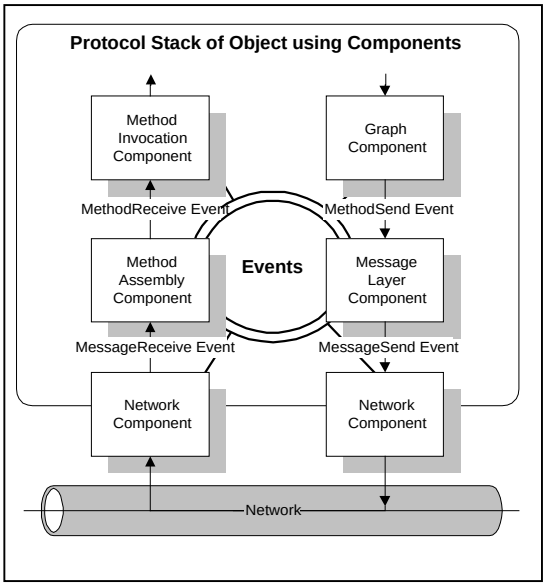


**Figure 2.** Sample protocol stack.

One of the primary applications of the RGE model is to implement a configurable protocol stack for Legion objects [30]. A striking feature of the protocol stack is that only a few events are employed. These events may be classified into three broad categories: message-related, method-related and object management-related events. These categories reflect the fact that Legion is an object-based system implemented at the low level over message passing. Table 1 describes several event kinds used in configuring the protocol stack.

| Category | Event Data | Event Kind | Description |
|---|---|---|---|
| Message-related events | Message and message headers | *MessageReceive* | Object has received a message |
| | | *MessageSend* | Object is sending a message |
| | | *MessageComplete* | Message has been successfully sent |
| | | *MessageError* | Error in sending message |
| Method-related events | Method signature, arguments, annotations | *MethodReceive* | Object has received a complete method invocation; all parameters have been received |
| | | *MethodReady* | A method has passed the security method access control check and is ready to be serviced |
| | | *MethodSend* | Object is invoking a method on a remote object |
| | | *MethodDone* | Object is done servicing a method |
| Object-management-related events | LOID of the object | *ObjectCreated* | An object has been created |
| | | *ObjectDeleted* | An object has been deleted |
| | State of the object. OPR organized in directory structure | *SaveState* | Saves the state of the object in its OPR (persistent storage) |
| | | *RestoreState* | Restores the state of the object from its OPR This event is raised upon object startup. |

**Table 1.** Events used to configure the protocol stack of Legion objects

Figure 2 illustrates the major components of the Legion protocol stack. When an object receives a message from the network, it announces a *MessageReceive* event. The `MethodAssemblyComponent` determines whether the received message is sufficient to form a complete method invocation (recall that in data flow multiple tokens/messages may be required to trigger a method execution). If the message results only in a partial method invocation, the object stores the message in an internal database. When the required messages arrive to complete the method invocation, a *MethodReceive* event is raised. At this point, the `MethodInvocationComponent`, stores the complete method in a database of ready methods. Then, a server loop may extract ready methods from the database and execute them. Once the method finishes executing, a *MethodDone* event is raised.

On the sending side, the `GraphComponent` announces a *MethodSend* event for each node in the graph that has the sender as a source of an input token. In turn, the `MessageLayerComponent` transforms each parameter into messages and announces a *MessageSend* event. Finally, the `NetworkComponent` sends messages over the network.

### 4.3 Exoevent Notification Model

The RGE model provides several ways of associating graphs and events. One way is for protocol writers to inspect and transform program graphs, or create and execute new graphs, within an event handler. Another more flexible approach is to associate graphs and events dynamically and execute a program graph when an event is raised, thereby enabling the run-time composition of functionality to objects. We call events associated with graphs *exoevents* to highlight the fact that raising such events may result in a set of remote method invocations. The benefit for object designers is that they need not anticipate all possible policies when building their objects.

Before showing an application of the exoevent notification model, we define the following terms: exoevent, exoevent interest, and exoevent interest set (EIS).

- Exoevent. An exoevent is a set of 3-tuple items <item-name, data-type, data-value>. The item-name field is a string to identify an item; the data-type specifies how to interpret the data-value field of an item. Items may be added or removed from an exoevent. Users may search for a specific item by using the name field as a key. By convention, all exoevents contain an item with item-name=”ExoEventType”. The data-type field is a string describing the type of exoevents. By convention, we classify exoevent types within broad categories and further divide them using a “:” to delineate subcategories, e.g., “Exception”, “Warning”, “Exception:Security”, “Exception:Security:Access Control”.
- Exoevent Interest. An exoevent interest is a 2-tuple <exoeventType, notificationGraph> that associates an exoevent type with a computation graph. The exoevent type specifies the kind of exoevent of interest. The notificationGraph is a first-class program graph and specifies a computation to be executed if a match is made between an exoevent and an exoevent interest.
- Exoevent Interest Set (EIS). An exoevent interest set is a set of exoevent interests. The EIS propagates to remote objects using implicit parameters.

Consider a server `S` used by multiple clients (Figure 3). By inserting the proper exoevent interest in its exoevent interest set, each client may specify its own exoevent propagation policy. Client $C_1$ specifies that exceptions propagate back to itself whereas $C_2$ specifies that warnings propagate to a third-party monitor object.

## 5 Incorporating Fault-tolerance Techniques into Applications using RGE

We present designs for encapsulating several well-known fault-tolerance techniques using the RGE model: two-phase commit distributed checkpointing, pessimistic method logging, passive replication and forward recovery. To encapsulate fault-tolerance techniques inside components, developers express their algorithms using graphs and events. Typically, this involves inserting handlers with the appropriate events or associating graphs with events.

Note that for these examples, we make the following assumptions:

- Objects are fail-stop, i.e., objects fail by halting and other objects may detect the failure.

- Objects are deterministic. Given a given sequence of input methods, objects will make the same state transitions.
- Objects always have access to stable storage via their OPR (Object Persistence Representation). Note that the OPR does not include the program counter or stack of the object as this information is not portable across heterogeneous architectures. This assumption can be relaxed using tools such as April that allow heterogeneous checkpoints [10].

Furthermore, we present only salient features of each technique due to space restrictions.

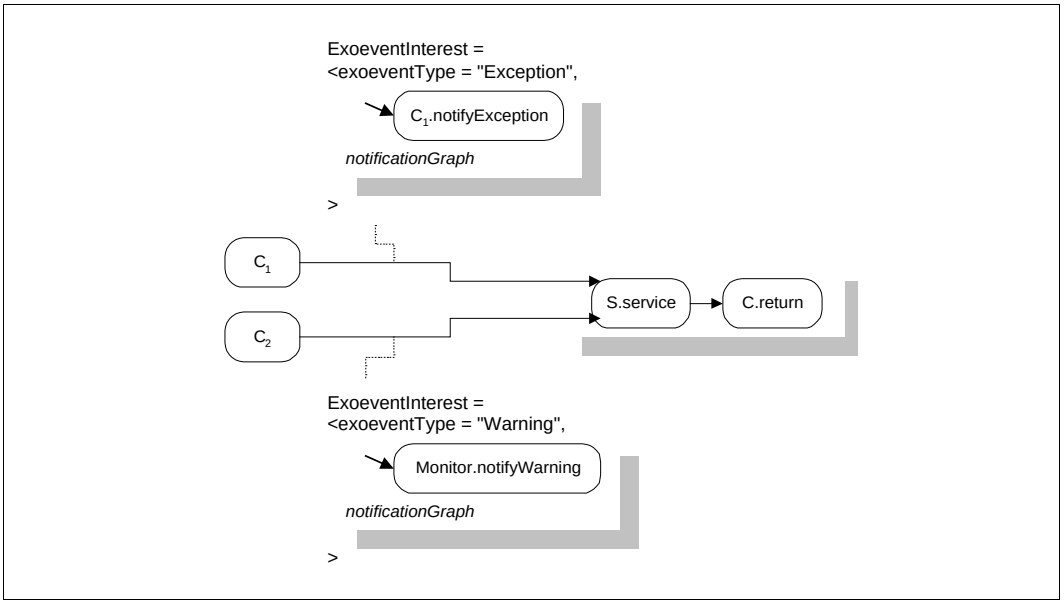


**Figure 3.** Clients $C_1$ and $C_2$ specify two different exoevent propagation policies. $C_1$ specifies that exceptions propagate back to it via the `notifyException()` method. $C_2$ specifies that warnings propagate to a third-party monitor object via the `notifyWarning()` method.

### 5.1 Two-Phase Commit Distributed Checkpointing (2PCDC)

A common method for ensuring the progress of long-running application is to checkpoint its state periodically on stable storage. Checkpoints may be viewed as "insurance policies" against failures—in the event of a failure, the application can be rolled back and restarted from its last checkpoint—thereby bounding the amount of lost work that must be recomputed. As with all insurance policies, there are choices and costs involved. Users may select from a variety of checkpointing algorithms, each providing a specified level of service for a given cost. Costs include the cost of the algorithm itself—memory, CPU, stable storage requirements, and run-time overhead—as well as the cost of implementing a given algorithm correctly. The RGE model directly addresses the latter cost: domain experts encapsulate fault-tolerance techniques into components that may be composed with user applications.

The basic idea behind a two-phase commit distributed checkpointing protocol is to ensure that either all objects in an application checkpoint or none do [21]. The set of local checkpoints taken must form a consistent global state—all methods received by an object must be recorded as having been sent. Two problems must be addressed to ensure a consistent global checkpoint: *lost* methods and *orphan* methods. Lost methods are methods that are marked as sent but not received, while orphan methods are methods marked as received but not sent (Figure 4). The algorithm presented here only seeks to prevent orphan methods; lost methods are assumed to be handled by the underlying communication channels.

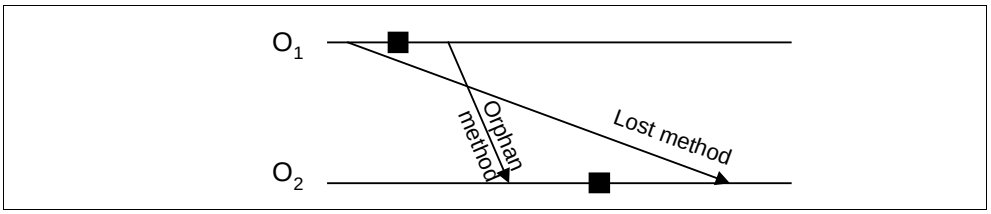


**Figure 4.** Two objects, $O_1$ and $O_2$, with local checkpoints (black boxes). Orphan methods result when a method is marked as received in one checkpoint ($O_2$) but not marked as sent in any other. Lost methods result when a method is marked as sent in one checkpoint ($O_1$) but not marked as received in any other.

The algorithm consists of two phases. In phase I, a coordinator requests participants to take a tentative checkpoint. If a participant rejects the request for any reasons, it replies `No`. Otherwise, the participant takes a tentative checkpoint, replies `Yes`, suspends communication with other objects, and awaits the coordinator's decision. If all participants reply `Yes`, the coordinator's decision is to commit the checkpoints, otherwise its decision is to abort the protocol. The coordinator's authoritative decision marks the end of the first phase. In phase II, the coordinator sends its decision to all participants. If the decision is `Yes`, participants commit the tentative checkpoint taken in the first phase to stable storage. Otherwise, participants may discard the tentative checkpoint previously taken.

| **Coordinator** | **Participants** |
|---|---|
| Phase I<br>requests participants to take tentative checkpoints<br>await all replies<br>if all replies = "Yes"<br>  decide Yes<br>else<br>  decide No | Phase I<br>if accept request<br>  take a tentative checkpoint<br>  reply Yes<br>  suspend communication<br>else<br>  reply No |
| Phase II<br>inform participants of decision | Phase II<br>if decision = "Yes"<br>  commit tentative checkpoint<br>else<br>  discard tentative checkpoint<br>resume communication |

**Table 1.** Overview of the Two-Phase Commit Distributed Checkpointing Algorithm

This basic algorithm may be extended in several ways. The coordinator can bound the amount of time that it waits for participants to reply. To handle a coordinator crash, the coordinator can save its decision onto stable storage at the end of phase I. The number of participants may be reduced by exploiting semantic information [21]. For the sake of brevity, we do not include these extensions in our mapping of the algorithm onto the RGE model, nor do we discuss the associated recovery algorithm.[3]

**Mapping onto the RGE model**

The algorithm is encapsulated using a `2PCDC` component. `2PCDC` adds the following methods to the participant's public interface, `void TakeTentativeCheckpoint()` and `void Decision()` so that the coordinator may invoke these methods.

In phase I, the `2PCDC` component takes a tentative checkpoint by raising a *SaveState* event. The default handlers for the event write the state of the object into the object's OPR. To suspend communication at the end of phase I and prevent orphan methods, we ensure that the next method serviced after `TakeTentativeCheckpoint()` is `Decision()`. `2PCDC` adds a handler, *AwaitDecision*, to the *MethodReceive* event. *AwaitDecision* intercepts all methods until the receipt of `Decision()`, at which point *AwaitDecision* announces a *MethodReady* event.

[3] Similarly to the checkpointing algorithm, the recovery algorithm uses a two phase-commit protocol

To commit the checkpoint in phase II, 2PCDC creates a new directory in the OPR, "/2PC-Commit", in which it writes the state of the object.

We ensure that there are no lost methods by preventing lost messages (recall that multiple messages may be needed to form a method invocation). Upon receipt of a message, an object raises a *MessageReceive* exoevent. The sender of the message registers its interest in *MessageReceive* using the Exoevent Notification Model (Section 4.3). If the invoker is not notified of *MessageReceive* in a timely manner, it retransmits the message. To handle duplicate messages, the invoking object appends a message identification number. The invoked object may then discard duplicates based on this number.

### 5.2 Pessimistic Method Logging (PML)

The two-phase commit distributed checkpointing algorithm requires objects to coordinate their local checkpoints to establish a consistent application global state. Further, during recovery, even objects that did not fail are potentially required to rollback their state. We now describe an adaptation of pessimistic message logging [31] for an object-based environment—pessimistic method logging (PML)—in which objects establish checkpoints and recover independently from one another.

In PML, objects checkpoint their state periodically and log received methods onto stable storage upon receipt before delivering the methods to the application layer. In the event of a failure, objects restart from their saved checkpoint and replay their log. Since objects are deterministic, replaying methods in the same order will produce the same execution (Figure 5).

PML is attractive due to its simple recovery characteristic—objects restart independently without the need for a costly coordination protocol. The disadvantage of PML is the high cost of saving methods onto stable storage. We do not discuss here techniques to reduce the overhead of pessimistic logging [17][18].

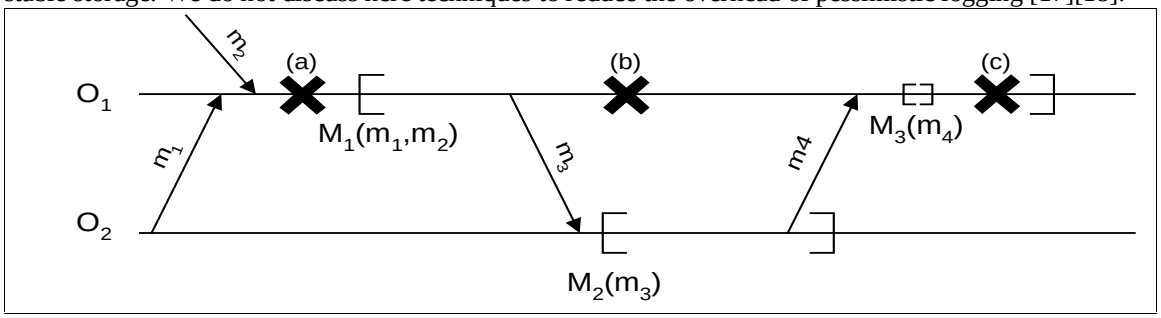


**Figure 5**. Multiple messages (lowercase m) may be needed to form a method invocation (uppercase M). Object $O_1$ crashes at (a), (b) or (c). If $O_1$ crashes at (a), $O_1$ replays messages $m_1$ and $m_2$ from the log. If $O_1$ crashes at (b), $O_1$ replays the method $M_1$ from the log. During recovery, message $m_4$ is retransmitted by $O_2$. If $O_1$ crashes at (c), $O_1$ replays method $M_1$ and message $m_4$ from the log.

**Mapping onto the RGE model**

The algorithm is encapsulated using a PML component. PML creates the following directories in the OPR of the object, "/MessageLog/" and "/MethodLog/". PML inserts a *LogMessage* handler with the *MessageReceive* event as well as a *LogMethod* handler with the *MethodReady* event. When an object receives a message, *LogMessage* writes it into the "/MessageLog/" directory. If the received message results in a full method invocation (recall that in our model, multiple messages may be needed to form a method), a *MethodReady* event is generated. *LogMethod* catches the event and writes the method into the "/MethodLog/" directory. To reclaim storage space, *LogMethod* also deletes the messages associated with the received method from "/MessageLog/". PML also inserts a *Restart* handler with the *RestoreState* event to ensure that PML is notified when an object restarts.

When communication is attempted on a crashed object, its class will restart it on an available host and restore its saved state. In the process, a *RestoreState* event is raised and caught by the *Restart* handler. *Restart* first replays the partial methods, i.e., messages, contained in "/MessageLog/". Then, it replays the methods contained in "/MethodLog/" before allowing normal processing to resume for the object. Replaying the method log may result in duplicate method invocations on remote objects. To prevent methods from executing multiple times, objects append to each message a unique message identifier so that receiving objects may discard duplicate entries.

To prevent lost methods and messages, PML uses the *MessageReceive* exoevent as described previously in Section 5.1. There are no orphan messages since all received messages are stored in the "/MessageLog/" directory in the OPR of objects.

### 5.3 Passive Replication

In passive replication, a primary object services method invocations. When the primary object finishes servicing a state-updating method it sends its new state to a backup object before replying to the caller. Upon failure of the primary, the backup takes over and services subsequent method invocations (Figure 6).

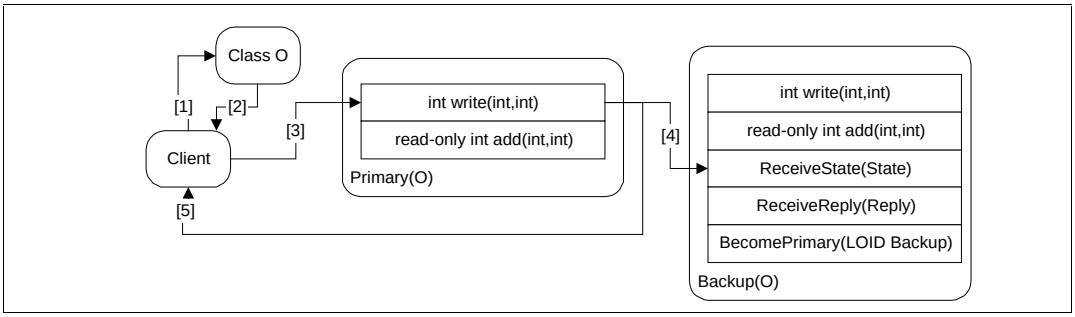


**Figure 6.** To communicate with O, a client first obtains a binding from Class O [1][2]. The client invokes a state-updating method, `write(),` on Primary(O) [3]. Before the result of `write()` is returned to the client [5], Primary(O) first forwards its state to the Backup [4]. If the primary fails, `Class O` invokes `BecomePrimary()` on the backup. Subsequent binding requests from clients will result in a binding to the new primary. Note that the methods `ReceiveState()`, `ReceiveReply()` and `BecomePrimary()`, are added transparently to the user code.

**Mapping onto the RGE model**

The passive replication algorithm is encapsulated inside of a PassiveReplication component. Inside the primary, PassiveReplication inserts a *SendStateToBackup* handler with the *MethodDone* event. If PassiveReplication is contained within the backup, it adds and exports the methods `BecomePrimary(LOID NewBackup)`, `ReceivePrimaryState(State S)`, and `ReceiveReply(Reply R)`.

At the primary, if the method serviced is state-updating, *SendStateToBackup* extracts the state of the object from its OPR and forwards it by invoking the method *SendStateToBackup* on the backup. *SendStateToBackup* determines whether a method is state-updating by inspecting the function signature. If the signature is not of the form "read-only return-type func(args…)", then the method is state-updating. If the method is non-state-updating, the primary sends only the reply value to the backup by invoking `ReceiveReply()`.

At the backup, PassiveReplication waits for the invocation of either the `BecomePrimary()` or `ReceivePrimaryState()` methods. If `BecomePrimary()` is invoked, the backup becomes primary and forwards its state to the new backup. Since the old primary can crash before sending the return value to the client, the new primary resends the last return value. Thus, clients may receive duplicate return values. We assume that clients can handle duplicate values.

When a binding request is issued for a crashed object, the default behavior is for the class of the object to restart the object on an available host, and return the new binding. Instead, the class now selects a replica as the new primary and invokes the `BecomePrimary()` method on the new primary, before returning the binding of the new primary. Passive replication results in faster recovery of crashed objects than the default algorithm as the backups are already active and ready to service methods.

### 5.4 Forward recovery

In forward recovery, applications do not rollback to a previously consistent state. Instead, they attempt to repair themselves so as to continue processing from a consistent state. By its nature, forward recovery is application-dependent and not as general as the backward-recovery methods discussed in the previous sections.

Our approach for supporting forward recovery is to use the exoevent notification model described in Section 4.3 in which the concepts of *raising* and *propagating* exceptions are decoupled. Thus, object writers need not specify exception propagation policies at design time.

**Mapping onto the RGE model**

If an object wishes to be notified of an exoevent raised by objects in its future call chain, it inserts an exoevent interest in its exoevent interest set. Consider a remote method invocation in which a client `C` invokes a method `service` on an object. To be notified of all exceptions raised by `S.service()`, `C` annotates its program graph with the exoevent interest shown in Figure 7. Since the exoeventType field is set to "`Exception`", all exceptions propagate back via the `notifyException` method on `C`.

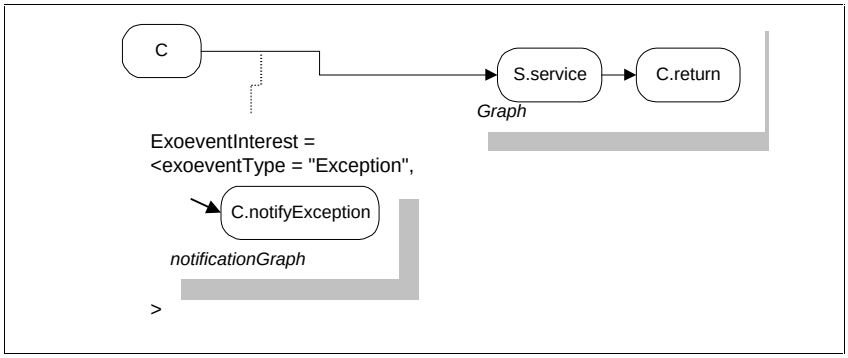


**Figure 7.** Client `C` specifies interest in exceptions raised by `S`.

## 6 Conclusion

To achieve our goal of alleviating the difficulty of writing *robust* metacomputing applications, we have presented a reflective model of computation, the Reflective Graph and Event model, for expressing fault-tolerance techniques inside reusable components and enabling the composition of such components with user applications. We have presented designs for mapping several well-known fault-tolerance techniques to the RGE model: two-phase commit distributed checkpointing, passive replication, pessimistic logging, and forward recovery.

The RGE model is implemented and deployed within the Legion metacomputing system. The forward recovery example has also been implemented and is in use. Future work consists of implementing and deploying the other examples described in this paper, as well as mapping other fault-tolerance techniques onto the RGE model.

## 7 References


[1] G. Agha and 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] A. Beguelin *et al.*, "HeNCE: Graphical Development Tools for Network-Based Concurrent Computing", *Proceedings SHPCC-92*, pp. 129-36, Williamsburg, VA, May 1992.

[5] 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.

[6] P. Charlton, "Self-Configurable Software Agents", *Advances in Object-Oriented Metalevel Architectures and Reflection*, CRC Press, pp. 103-127, 1996.

[7] 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.

[8] 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.

[9] I. Foster and C. Kesselman, "Globus: A metacomputing infrastructure toolkit", *International Journal of Supercomputing Applications*, 1997.

[10] A. Ferrari, "Process Introspection: A Checkpoint Mechanism for High Performance Heterogeneous Distributed Systems", *Department of Computer Science Technical Report CS-96-15*, University of Virginia, October 1996.
[11] A. S. Grimshaw, "The Legion vision of a worldwide virtual computer", *Communications of the ACM*, 40:1, pp. 39-45, January 1997.
[12] A. S. Grimshaw, A. Ferrari and E. West, "Mentat", *Parallel Programming Using C++*, The MIT Press, Cambridge, Massachusetts, pp. 383-427, 1996.
[13] A. S. Grimshaw *et al.*, "Architectural Support for Extensibility and Autonomy in Wide-Area Distributed Object Systems ", *Department of Compter Science Technical Report CS-98-12*, University of Virginia, June 1998.
[14] A. S. Grimshaw, J. B. Weissman and T. Strayer, "Portable Run-Time Support for Dynamic Object-Oriented Parallel Processing", *ACM Transactions on Computer Systems*, Vol. 14, Num. 2, 1996.
[15] M. Hayden, "The Ensemble System", *Cornell University Technical Report*, TR98-1662, January 1998.
[16] Y. Honda and M. Tokoro, "Soft Real-Time Programming through Reflection", *Proceedings of the International Workshop on New Models for Software Architecture: Reflection and Metalevel Architecture*, pp. 12-23, 1992.
[17] P. Jalote, "Fault Tolerance in Distributed Systems", Prenctice Hall, 1994.
[18] D. B. Johnson and W. Zwaenepoel, "Sender-Based Message Logging", *The Seventeenth Symposium on Fault-Tolerant Computing (FTCS-17)*, pp. 14-19, 1987.
[19] G. Kiczales, J. D. Rivieres and D. G. Bobrow, "The Art of the Metaobject Protocol", MIT Press, 1991.
[20] A. H. Lee and J. L. Zachary, "Reflections on metaprogramming", *IEEE Transactions on Software Engineering*, vol. 21, pp. 883-892, November 1995.
[21] R. Koo and S. Toueg, "Checkpointing and Rollback-Recovery for Distributed Systems", *IEEE Transactions on Software Engineering*, pp. 23-31, January 1987.
[22] 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.
[23] A. Nguyen-Tuong *et al.*, "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. 2-11, 1996.
[24] A. Nguyen-Tuong *et al.*, "Using Reflection for Flexibility and Extensibility in a Metacomputing Environment", *Technical Report CS-98-33*, Department of Computer Science, University of Virginia, 1998.
[25] OMG, "The Common Object Request Broker: Architecture and Specification", *OMG*, 1995.
[26] 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.
[27] J. A. Stankovic, S. H. Son and 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.
[28] R. J. Stroud and Z. Wu, "Using Metaobject Protocols to Satisfy Non-Functional Requirements", *Advances in Object-Oriented Metalevel Architectures and Reflection*, Chapter 3, CRC Press, pp. 31-52, 1996.
[29] Sun Microsystems, "JavaBeans™", *http://www.javasoft.com/beans/*, September 1998.
[30] 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.
[31] Y. Huang and C. Kintala, "A software fault tolerance platform", *Practical Reusable Software*, Ed. B. Krishnamurthy, John Wiley & Sons, pp. 223-245, 1995.