Egida: An Extensible Toolkit For Low-overhead Fault-Tolerance

Sriram Rao, **Lorenzo Alvisi**, and **Harrick M. Vin**

Department of Computer Sciences
The University of Texas at Austin
Taylor Hall 2.124, Austin, Texas 78712-1188, USA
E-mail: {sriram,lorenzo,vin}@cs.utexas.edu, Telephone: (512) 471-9792, Fax: (512) 471-8885
URL: http://www.cs.utexas.edu/users/{sriram,lorenzo,vin}

Abstract

We discuss the design and implementation of *Egida,* an object-oriented toolkit designed to support transparent rollback-recovery. Egida exports a simple specification language that can be used to express arbitrary rollback recovery protocols. From this specification, Egida automatically synthesizes an implementation of the specified protocol by glueing together the appropriate objects from an available library of "building blocks". Egida is extensible and facilitates rapid implementation of rollback recovery protocols with minimal programming effort. We have integrated Egida with the MPICH implementation of the MPI standard. Existing MPI applications can take advantage of Egida without any modifications: fault-tolerance is achieved transparently—all that is needed is a simple re-link of the MPI application with Egida. We demonstrate Egida's versatility both as a testbed as well as an environment for developing new protocols by generating a few message logging protocols and evaluating their performance with a set of NAS benchmarks on a network of workstations.

1 Introduction

Building reliable distributed application is complex. Application developers not only have to worry about the intricacies of their applications, but also have to understand the subtleties of distributed fault-tolerance. Over the past decade, toolkits such as Isis [2], Horus [27], and Transis [8] have addressed this issue by providing higher level primitives for implementing fault-tolerant broadcasting and group communication. These primitives are well-suited for implementing fault-tolerance protocols based on active replication, desirable for mission-critical applications that require the highest degree of availability and the capability to tolerate arbitrary failures.

As distributed computing becomes commonplace, however, a growing number of non mission-critical applications are emerging. For instance, parallel scientific applications that used to be run on supercomputers can be executed on a distributed cluster of servers. For these applications, fault-tolerance is highly desirable, but only if it can be provided with low-overhead in terms of dedicated resources and execution time. Log-based rollback recovery protocols—such as checkpointing and message logging—provide an attractive low-overhead solution for building non-critical distributed applications that can tolerate crash failures.

In this paper, we discuss the design and implementation of *Egida*, an object-oriented toolkit designed to support transparent rollback recovery for low-overhead fault-tolerance. Rather than providing monolithic implementations of a set of protocols, we build Egida around a library of objects that implement a set of functionalities that are at the core of all log-based rollback recovery protocols and define a grammar for configuring protocols from the library of objects. Our approach has several advantages. First, it promotes extensibility and flexibility by allowing multiple implementations of each of the core functionalities. Second, it facilitates rapid implementation of rollback recovery protocols with minimal programming effort by gluing together objects from the available library of building blocks. This enables application developers to experiment with different protocols and then use the one that closely matches the requirements of their application. Finally, Egida enables designers of fault-tolerance protocols to develop new rollback recovery protocols by combining different implementations of the core functionalities in novel ways.

Our approach of configuring protocols from basic building blocks is similar to the one used in other systems—for instance, in xkernel [14] for networking protocols, and in Horus [27] and Cactus [12] for distributed computing and group membership protocols. However, to our knowledge, Egida is the first application of this approach to rollback recovery protocols.

We have integrated Egida with the MPICH implementation of the Message Passing Interface (MPI) standard [30]. This enables existing MPI applications to take advantage of Egida without any modifications. Conversely, the performance of rollback recovery protocols can now be evaluated using a large set of demanding applications.

This paper is organized as follows. In Section 2, we provide a brief overview of log-based rollback recovery protocols. In Section 3, we identify the set functionalities that are at the core of all log-based rollback recovery protocols, and present a grammar for synthesizing these protocols from its core components. Section 4 describes the architecture of Egida. Section 5 describes the implementation of Egida, its integration with the MPICH library [10, 11], and an evaluation of a set of rollback recovery protocols using the NAS benchmark suite [6]. Section 6 presents the related work, and finally, Section 7 summarizes our contributions.

2 Flavors of Log-based Rollback Recovery

Log-based rollback recovery protocols—such as checkpointing and message logging—provide a low-overhead solution for building distributed applications that tolerate crash failures. These protocols come in several flavors. Checkpoints can be independent, coordinated, or induced by specific patterns of communication. Logging can be pessimistic, optimistic, or causal. Pessimistic protocols [3, 15, 23] allow processes to communicate only from recoverable states. These protocols enforce this condition by synchronously logging to stable storage any information critical for recovery before letting processes communicate. Optimistic protocols [7, 16, 17, 29, 32, 34] allow processes to communicate with other processes even from states that are not yet recoverable. These protocols guarantee that these states will eventually become recoverable, but only if no failures occur. Failures may indeed render some states permanently unrecoverable, forcing the rollback of any process that depends on such states. Causal protocols [1, 9] weaken the condition imposed by pessimistic protocols and allow the possibility that a state from which a process communicates may become unrecoverable because of a failure, but only if no correct process depends on that state. This is enforced by appending to all communication the information necessary to recover the state from which the communication originates. This information is replicated in the volatile memory of the processes that causally depend [18] on the originating state. Hence, causal protocols never roll back correct processes, but do not require synchronous writes to stable storage.

3 Deconstructing Log-Based Rollback-Recovery Protocols

The diversity of rollback-recovery protocols reflects the heterogeneity in the requirements of applications. For instance, applications that don't interact frequently with the external environment and can tolerate rollback of correct processes may use pure checkpointing or optimistic protocols. Pessimistic and causal protocols are better suited for applications in which communication with the environment is frequent and rollbacks are unacceptable; pessimistic protocols are preferred when fast and simple recovery is required. Applications that instead demand minimal overhead during failure-free executions would use causal or optimistic protocols. A closer look at these protocols, however, reveals that behind this diversity lies a simple event-driven structure that all these protocols share and that all protocols are interested in the same set of "relevant" events. In this section we identify such events and present a language that can be used to specify a protocol in terms of which actions it takes in response to each of these events.

3.1 Log-based Rollback-Recovery Protocols as Event-Driven Programs

There are five types of events that are relevant to all log-based rollback recovery protocols. These are: (1) non-deterministic events, (2) dependency-generating events, (3) output-commit events, (4) checkpointing events, and (5) failure-detection events.

1. **Non-deterministic events**: A non-deterministic event is an event whose outcome may change for different executions of the same program. For example, a message deliver event is often non-deterministic since its outcome depends on many factors, including process scheduling, routing, and flow control. Thus, a recovering process may not produce the same run upon recovery even if the same set of messages are sent to it.

   If a non-deterministic event cannot be replayed during the recovery of a process, all correct processes whose state depends on that unrecoverable event must be rolled back. These processes are called *orphans*. The information necessary to reproduce the results of the non-deterministic events is called the event's *determinant*. Restoring the system to a consistent state while limiting the extent of rollbacks depends on the ability of the rollback-recovery protocol to replay the determinants of the non-deterministic events executed by the failed processes before crashing. Hence, determinants must be saved to stable storage so that they are available during recovery. In practice, different protocols choose different ways to log determinants and rely on different implementations of stable storage.

2. **Dependency-generating events**: These events can increase the number of processes that depend on the non-deterministic events executed by a process. For example, in message-passing applications, a message-send event is a dependency-generating event since it can create dependencies between non-deterministic events executed by the sender and the state of the receiver. If the sender fails and any of those non-deterministic events is unrecoverable, then the receiver becomes an orphan.

   To enable orphan detection and appropriate rollback, log-based recovery protocols typically append to application messages information (such as logical clocks [18] or vector clocks [20]) that tracks these dependencies. In addition, to speedup the replay of non-deterministic events, it may be desirable to log some information on executing a dependency-generating event. For instance, in sender-based message-logging protocols processes log the content of each message they send.

3. **Output-commit events**: These events can make the external environment depend on the non-deterministic events executed by a process.

Since it is not reasonable to expect the external environment to roll back in response to a process crash, log-based rollback recovery protocols invoke *output commit* procedures that write synchronously to stable storage the determinants of all non-deterministic events that precede the communication with the external environment, thus making them recoverable. Depending on how determinants are logged, this procedure may require coordination among processes.

4. **Checkpointing events**: These events instruct the protocols to write to stable storage the state of one or more processes. These event can either be transparent to the application (e.g. a signal from a timer indicating that some pre-determined time has elapsed since the last checkpoint) or be explicitly driven by the application.

   Checkpoints can be coordinated, independent, or communication induced. Processes may checkpoint their state incrementally, and checkpoints can be saved to stable storage either synchronously or asynchronously.

5. **Failure-detection events**: These events are generated on detecting the failure of one or more processes.

   In response to these events, faulty processes must be restarted, restored to a previously checkpointed state and rolled forward. In turn, correct orphan processes may need to be detected and rolled back. Different protocols implement different strategies to collect the determinants to be replayed during recovery and to detect and rollback orphans.

Implementing a specific protocol therefore amounts to selecting the set of actions performed in response to each relevant event. We now show how a simple language can be used to specify these choices. In Section 4 we show how protocols can be automatically synthesized starting from these specifications.

3.2 Specifying Design Choices in Rollback-Recovery Protocols

Figure 1 shows a simple language that can be used to specify rollback-recovery protocols. The language reflects the discussion of the previous section. A protocol is defined in terms the actions it takes in response to non-deterministic events, dependency generating events, output commit events, checkpointing events and failure-detection events. To define a protocol completely, it is necessary to instantiate a set of variables which specify, for instance, the set of non-deterministic events, the form of their determinant, the implementation of stable storage, etc. Figure 2 shows a set of representative instantiations of these variables used in several existing protocols.

Figure 3 illustrates how the language can be used to specify Bacon, Strom, and Yemini's sender-based pessimistic protocol [33], Damani and Garg's optimistic protocol [7], and a hybrid protocol we have recently developed that combines causal logging with asynchronous receiver-based logging to stable storage in order to speed up crash recovery [26]. Our language, however, can be used for more than just specifying existing recovery protocols. Once deconstructing rollback-recovery protocols becomes possible, it is then easy to reassemble the fundamental components in novel ways to obtain new protocols. An example is given in Figure 4 , which shows a specification of a sender-based pessimistic message-logging protocol that tolerates f concurrent failures and implements stable storage by replicating data in the volatile memory of processes. In this protocol, which to our knowledge has not appeared in the literature, a process executing a non-deterministic event broadcasts the corresponding determinant to the other processes and does not send any application message until it has received at least f acknowledgments to its broadcast.

4 The Architecture of Egida

Figure 1 defines the structure of log-based rollback recovery protocols, while the variables in Figure 2 identify the building blocks which when incorporated into the protocol structure yield different rollback recovery protocols. Egida instantiates these building blocks in a modular, object-oriented architecture. We begin this section by describing the functionality of each module and the interfaces it exports. We then explain how these modules are invoked in response to events and how they can be composed to synthesize rollback-recovery protocols.

4.1 Module Definitions and Interfaces

Egida defines the following modules:

EventHandler: Handles the five types of relevant events defined in Section 3. The methods exported by this module are event-dependent.

Determinant: Creates the determinants for each non-deterministic event. It exports a single method, *CreateDeterminant*.

Protocol	:=	⟨`non-det-event-stmt`⟩*
		⟨`output-commit-event-stmt`⟩*
		⟨`dep-gen-event-stmt`⟩*
		⟨`ckpt-stmt`⟩$_{opt}$
		⟨`recovery-stmt`⟩$_{opt}$
⟨`non-det-event-stmt`⟩	:=	⟨event⟩ :
		determinant : ⟨determinant-structure⟩
		⟨*Log* ⟨how-to-log⟩ *on* ⟨stable-storage⟩⟩$_{opt}$
⟨`output-commit-event-stmt`⟩	:=	⟨output-commit-proto⟩ *output commit on* ⟨`event-list`⟩
⟨`event-list`⟩	:=	⟨event⟩ | ⟨event⟩, ⟨`event-list`⟩
⟨`dep-gen-event-stmt`⟩	:=	⟨event⟩ :
		⟨`piggyback-stmt`⟩ | ⟨`logging-stmt`⟩ |
		⟨`piggyback-stmt`⟩⟨`logging-stmt`⟩
⟨`logging-stmt`⟩	:=	*Log* ⟨`event-info-list`⟩ ⟨how-to-log⟩ *on* ⟨where-to-log⟩
⟨`event-info-list`⟩	:=	⟨event-info⟩ | ⟨event-info⟩, ⟨`event-info-list`⟩
⟨`piggyback-stmt`⟩	:=	*Piggyback* ⟨`pb-value-list`⟩
⟨`pb-value-list`⟩	:=	⟨pb-value⟩ | ⟨pb-value⟩, ⟨`pb-value-list`⟩
⟨`ckpt-stmt`⟩	:=	⟨ckpt-proto⟩ *checkpoint* ⟨how-to-ckpt⟩ *on* ⟨stable-storage⟩
		Implementation : ⟨ckpt-impl⟩
⟨`recovery-stmt`⟩	:=	⟨`recovery-get-det-stmt`⟩$_{opt}$
		⟨`orphan-detection-stmt`⟩$_{opt}$
		⟨`rollback-stmt`⟩$_{opt}$
⟨`recovery-get-det-stmt`⟩	:=	*Protocol to retrieve determinants* : ⟨recovery-get-det-proto⟩
⟨`orphan-detection-stmt`⟩	:=	*Protocol to detect orphans* : ⟨orphan-detection-proto⟩
⟨`rollback-stmt`⟩	:=	*On rollback* :
		⟨`rb-logging-stmt`⟩ | ⟨`rb-ckpt-stmt`⟩
⟨`rb-logging-stmt`⟩	:=	*Log* ⟨`event-info-list`⟩ *on* ⟨stable-storage⟩
⟨`rb-ckpt-stmt`⟩	:=	*Take independent checkpoint on* ⟨stable-storage⟩

Notational Conventions :	
⟨`x`⟩	means that `x` is a production rule
⟨y⟩	means that y is a variable that has to be mapped to a specific value
⟨z⟩$_{opt}$	means that ⟨z⟩ is an optional statement
a	means that *a* is a keyword

Figure 1: A grammar for specifying rollback-recovery protocols.

⟨event⟩	:=	send | receive | read | write
⟨determinant-structure⟩	:=	{source, sesn, dest, desn}
⟨output-commit-proto⟩	:=	independent | co-ordinated
⟨event-info⟩	:=	determinant | message
⟨how-to-log⟩	:=	synchronously | asynchronously
⟨where-to-log⟩	:=	⟨stable-storage⟩ | ⟨volatile-storage⟩
⟨volatile-storage⟩	:=	local disk | volatile memory of self
⟨stable-storage⟩	:=	local disk | NFS disk | volatile memory of processes
⟨pb-value⟩	:=	vector clock | logical clock |
		determinants⟨, ⟨dependency matrix | stability matrix | stability vector⟩⟩$_{opt}$
⟨ckpt-proto⟩	:=	independent | co-ordinated | communication-induced
⟨how-to-ckpt⟩	:=	synchronously | asynchronously
⟨ckpt-impl⟩	:=	full | incremental
⟨recovery-get-det-proto⟩	:=	centralized | distributed
⟨orphan-detection-proto⟩	:=	broadcast logical clock | exchange vector clock

Figure 2: Representative instantiations for the variables defined in the grammar shown in Figure 1.

HowToOutputCommit**:** Determines how to save the information necessary to recover the system to the state in which an output commit event is executed. It exports a single method, *OutputCommit*.

LogEventDeterminant**:** Saves the determinants of non-deterministic events to stable storage. It exports four methods: (1) *Log*, (2) *Retrieve*, (3) *Flush*, and (4) *GarbageCollect*.

LogEventInfo**:** Logs data associated with an event. It exports the same interfaces as the LogEventDeterminant module.

HowToLog**:** Determines how logged information is accessed. It exports two methods, *Read* and *Write*.

WhereToLog**:** Directs requests for accessing the logs to the appropriate form of storage. It exports the same methods as HowToLog.

StableStorage**:** Implements stable storage for determinants and data associated with an event. It also exports two methods, *Read* and *Write*.

VolatileStorage**:** Implements volatile storage for determinants and data associated with an event. It exports the same methods as StableStorage.

Specifying Bacon, Strom and Yemini's sender-based pessimistic message logging protocol

/∗ `comment:` ⟨non-det-event-stmt⟩ ∗/
receive:
    determinant : {source, sesn, dest, desn}
    Log determinant synchronously on NFS disk
/∗`comment:` ⟨dep-gen-event-stmt⟩} ∗/
send:
    Log message synchronously on volatile memory of self
/∗ `comment:` ⟨ckpt-stmt⟩} ∗/
Independent checkpoint asynchronously on NFS disk
Implementation : incremental

Specifying Damani and Garg's optimistic message logging protocol

/∗ `comment:` ⟨non-det-event-stmt⟩ ∗/
receive:
    determinant : {source, sesn, dest, desn}
    Log determinant asynchronously on NFS disk
{`comment:` ⟨output-commmit-event-stmt⟩}
Co-ordinated output commit on write
/∗ `comment:` ⟨dep-gen-event-stmt⟩ ∗/
send:
    Piggyback vector clock
    Log message synchronously on volatile memory of self
/∗ `comment:` ⟨recovery-stmt⟩ ∗/
Protocol to detect orphans : broadcast logical clock
On rollback: Log determinant, message on NFS disk

Specifying hybrid causal message logging protocol

/∗ `comment:` ⟨non-det-event-stmt⟩ ∗/
receive:
    determinant : {source, sesn, dest, desn}
    Log determinant, data asynchronously on NFS disk
/∗ `comment:` ⟨dep-gen-event-stmt⟩ ∗/
send:
    Piggyback determinants
    Log message synchronously on volatile memory of self
/∗ `comment:` ⟨recovery-stmt⟩ ∗/
Protocol to retrieve determinants : centralized

Figure 3: Example specifications of existing protocols

```
/* comment: ⟨non-det-event-stmt⟩ */
receive:
    determinant : {source, sesn, dest, desn}
    Log determinant synchronously on volatile memory of processes
/* comment: ⟨dep-gen-event-stmt⟩ */
send:
    Log message synchronously on volatile memory of self
/* comment: ⟨ckpt-stmt⟩ */
Independent checkpoint asynchronously on NFS disk
Implementation : incremental
```

Figure 4: Specification of a novel sender-based pessimistic protocol

PiggybackLogging**:** Piggybacks onto application messages the information necessary to track dependencies among process states and, in the case of causal logging, appends the non-stable determinants logged by the message sender. It also processes, and if appropriate logs the piggybacked information. This module exports two methods: *GetPiggyback* and *ProcessPiggyback*.

PiggybackCheckpointing**:** Piggybacks the information necessary to track the dependencies among checkpoints. It also processes piggybacked information, and if appropriate triggers a checkpoint. It exports the same methods as PiggybackLogging.

Checkpoint**:** Implements the checkpointing protocol. It exports two methods, *TakeCheckpoint* and *RestoreFromCheckpoint*.

HowToCheckpoint**:** Determines how the checkpoint information is accessed. It exports two methods, *Read* and *Write*.

CollectDeterminants**:** Collects determinants during recovery. It exports a single method, *RetrieveDeterminants*.

OrphanDetection**:** Implements the protocol for orphan detection. It exports a single method, *DetectOrphans*.

4.2 Module Invocation

Figure 5 shows the dependency graph for the modules introduced in Section 4.1. In addition to the modules discussed in Section 4.1, it shows four other modules:

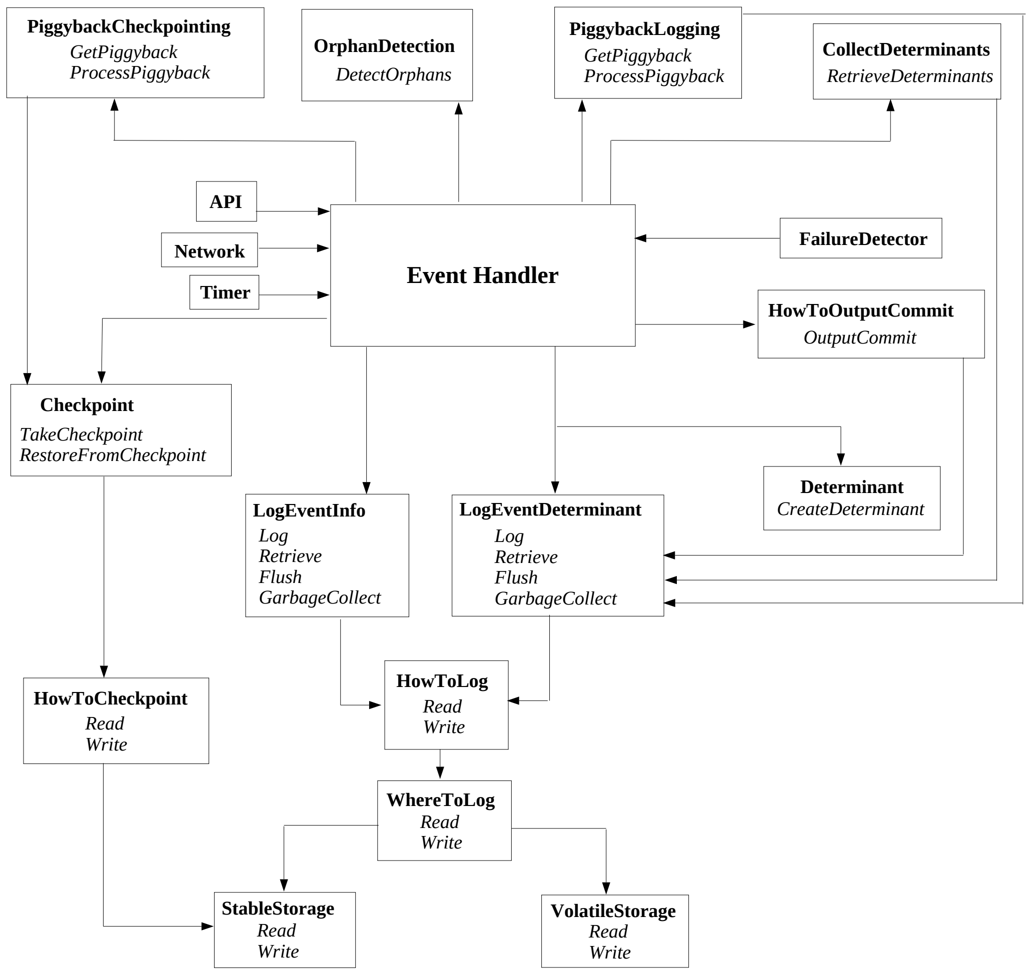

Figure 5: The architecture of Egida

1. An API module that intercepts application-level calls to relevant events and invokes the appropriate handler exported by the EventHandler.

2. A Timer module that enables Egida to specify the intervals at which timer interrupts need to be generated to perform periodic actions, such as checkpointing and flushing of volatile logs to stable storage.

3. A FailureDetector module that implements a protocol for detecting process crashes and triggers the appropriate recovery mechanisms through the EventHandler.

4. A Network module that invokes a method of EventHandler to process incoming messages.

To illustrate the interactions among Egida's module, we present two examples that show how this dependency graph is traversed in response to specific events.

Example 1 *Handling a Send event for a causal logging protocol (*see [1]*):*

The API module intercepts the Send event and invokes the corresponding handler in EventHandler. The handler in this example performs three tasks: (1) it saves in a volatile log kept by sender the content of the message being sent; (2) it retrieves the set of non-stable determinants from the log kept by the server; (3) it piggybacks these determinants on the application message and sends the message to the destination.

1. To save the content of the message in the sender's volatile log, the handler invokes the *Log* method of LogEventInfo. *Log* in turn invokes the *Write* method of HowToLog, which invokes the *Write* of WhereToLog, which finally invokes the *Write* method exported by VolatileStorage.

2. To obtain the determinants to piggyback, the handler invokes the *GetPiggyback* method of PiggybackLogging. This method identifies the non-stable determinants and invokes the *Retrieve* method of LogEventDeterminant. *Retrieve* in turn invokes the *Read* method of HowToLog, which invokes the *Read* of WhereToLog, which finally invokes the *Read* of VolatileStorage and, if necessary, of StableStorage. These methods return the requested determinants which are then passed up through the call chain back to *GetPiggyback*.

3. The event handler then piggybacks the determinants to the application message and invokes the send function in the transport layer[1]. ■

[1]As we will see in Section 5, in our current implementation the piggybacked determinants are sent in a separate message in order to avoid unnecessary data copying.

Example 2 *Handling a Send event to the external environment for the optimistic protocol of Damani and Garg (*see [7]*):*

The API module intercepts the send event and invokes the corresponding handler in EventHandler. The handler in this example invokes the output commit procedure prior to sending the message. The output commit procedure in turn performs two tasks: (1) it flushes to stable storage the determinants of all the non-deterministic events executed by the process prior to the send to the external environment, thereby making them recoverable; and (2) it initiates a co-ordinated output commit operation to ensure that all the non-deterministic events that causally precede the send are recoverable.

1. To perform output commit, the handler invokes the *OutputCommit* method of HowToOutputCommit. The *OutputCommit* methods performs the following two tasks:

   (a) To perform a local output commit, *OutputCommit* invokes the *Flush* method of LogEventDeterminant. *Flush* then invokes the *Write* method of HowToLog, which invokes the *Write* of WhereToLog, which finally invokes the *Write* method exported by StableStorage.

   (b) To perform a co-ordinated output commit, *OutputCommit* sends a message to all other processes requesting an output commit. On receiving this message, the Network module invokes the appropriate handler in EventHandler, which in turn invokes the *OutputCommit* method of HowToOutputCommit[2]. The *OutputCommit* method then invokes the same sequence of methods as in (a).

2. The handler sends the application message to the external environment. ■

4.3 Synthesizing Protocols through Module Composition

Egida allows the co-existence of multiple implementations for each of the modules introduced in Section 4.1. Hence, to synthesize a protocol, one must select a specific implementation of each module. Egida maintains a binding between the values for the variables on the left hand side of Figure 2 and their corresponding implementations. Therefore, synthesizing a protocol requires processing the specification along with this binding information to initialize the modules to their appropriate implementations. It is easy to extend Egida to recognize new implementations of modules; as more implementations of a module functionality

[2]Note that the *OutputCommit* method exported by the instance of the HowToOutputCommit module invoked by the handler for processing network messages performs only a local output commit.

become available, the specification language can be extended by simply (1) adding a new value to the right hand side of the corresponding variable in Figure 2, and (2) registering with Egida the binding between the new value and its corresponding implementation.

5 Implementation and Experience

5.1 Implementation Status

Egida contains multiple implementations for each module introduced in Section 4.1. Each module is defined as a C++ class. The classes we have implemented so far correspond to the values on the right hand side of Figure 2.

To make Egida available to a large number of applications, we have integrated Egida with MPICH [10, 11], a freely available, portable implementation of the MPI (Message Passing Interface) standard 1.1. MPICH has a two-layer architecture: the upper layer exports MPI's application programming interface (API), while the lower layer contains platform-specific message-passing libraries. For our testbed environment consisting of a network of workstations connected by an Ethernet, MPICH's lower layer is implemented using the p4 library [4].

To integrate Egida with MPICH, we replaced in the MPICH's upper layer all the send and receive calls to p4 with corresponding calls to Egida's API; modules in Egida in turn invoke the message passing operations of p4. To prevent data copying, MPICH messages and the information piggybacked by Egida are transmitted separately. We also modified p4 to handle socket errors that occur whenever processes fail and to allow a recovering process to establish socket connections with the surviving processes. Our implementation provides transparent fault-tolerance to MPI applications—only a simple re-link of the application with Egida and our modified MPICH library is necessary.

5.2 Evaluating Message Logging Protocols

We have used Egida to compare the performance of five message logging protocols—the four protocols described in Figures 3 and 4 and a receiver-based pessimistic protocol. We compare the performance of these protocols with respect to: (1) the overhead imposed by these protocols during failure-free executions, and (2) the time taken to restore a process to its pre-crashed state.

Application	NPB Specific Info	Per Process Avg. Message Rate		Exec. Time (sec.)
		Messages/sec.	KB/sec.	
BT	Class A	7	352.3	1530
CG	Class B	15	752.0	1516
LU	Class B	26	150.3	4017
SP	Class B	7	540.9	4530

Table 1: Characteristics of the benchmarks used in the experiments

5.2.1 Experimental Setting

To compare the performance of the five message logging protocols, we use four long-running compute-intensive applications from the NPB2.3 benchmark suite developed by NASA's Numerical Aerodynamic Simulation program [6]. These benchmarks are derived from computational fluid dynamics codes; the characteristics of these benchmarks are shown in Table 1.

We conducted our experiments using a cluster of four Pentium-based workstations connected by a lightly-loaded 100Mb/s Ethernet. Each workstation contains two 300-MHz Pentium-2 processors, 512 MBytes of memory, 4 GB disk, and runs Solaris 2.6. For all the experiments, each workstation executes a single application process. Stable storage is provided by an NFS file server. Checkpoints are taken five minutes apart and are saved to stable storage asynchronously. We exploit the iterative nature of the applications, and induce failures after a process has completed a pre-determined number of iterations. This ensures that the amount of lost computation that has to be reproduced by a recovering process is the same for all protocols.

The NPB benchmarks, MPICH library, and Egida , respectively, were compiled using SUN's f77, C, and C++ compilers with the optimization flag "–O". The results reported in this paper are with a 95% confidence interval obtained by averaging the results of at least five runs.

5.2.2 Performance Results

Failure-free Performance

Table 2 shows the failure-free execution times as well as the overhead—relative to the application execution times reported in Table 1—imposed by the three pessimistic protocols. Table 3 shows the corresponding results for the optimistic and hybrid causal protocols.

As expected, the pessimistic protocols impose a higher failure-free overhead as compared to the optimistic and hybrid causal protocols, and among the pessimistic protocols, the receiver-based protocol imposes the

Application	**Receiver-based**		**Sender-based (Disk)**		**Sender-based (Vol. Memory)**	
	Exec. Time (sec.)	% Overhead	**Exec. Time** (sec.)	% Overhead	**Exec. Time** (sec.)	% Overhead
BT	2068	33.1%	1692	8.9%	1602	3.1%
CG	3157	109.4%	1888	25.2%	1790	18.7%
LU	8599	114.1%	5960	48.4%	7243	80.3%
SP	5795	27.9%	4986	10.1%	5240	15.7%

Table 2: Failure-free results for the three variations of pessimistic message logging protocols with a checkpoint interval of 5 min.

Application	**Optimistic**		**Hybrid Causal**	
	Exec. Time (sec.)	% Overhead	**Exec. Time** (sec.)	% Overhead
BT	1573	1.2%	1580	1.7%
CG	1720	14.1%	1733	14.9%
LU	4345	8.2%	4644	15.6%
SP	4626	2.1%	4636	2.3%

Table 3: Failure-free results for optimistic and hybrid causal message logging protocols with a checkpoint interval of 5 min. In both protocols, a process flushes its volatile logs to disk once every minute.

highest overhead. The difference in the performance between the two sender-based pessimistic protocols is application-dependent. The performance of the protocol that implements stable storage in volatile memory depends heavily on the responsiveness with which acknowledgments are returned. This, however, depends on the message frequency and the size of messages exchanged by application processes.

The optimistic and hybrid protocols perform comparably during failure-free execution, except for the **LU** application where optimistic significantly outperforms hybrid. This is because the overhead of processing the piggybacked is higher for causal protocols than for optimistic; applications, such as **LU**, with high message frequency amplify this difference.

Recovery Performance

Table 4 compares the time taken by the five protocols to restore a process to its pre-crashed state for the **CG** benchmark with different number of concurrent failures.

When $f = 1$, protocols that never rollback correct processes provide fast crash recovery. When $f > 1$, the dominant factor in determining performance is *where* the recovery information is logged:

f	Pessimistic			Optimistic	Hybrid Causal
	Receiver-based	**Sender-based (Disk)**	**Sender-based (Vol. Memory)**		
	(sec.)	(sec.)	(sec.)	(sec.)	(sec.)
1	146.2	168.8	170.2	160.0	169.4
2	146.9	176.4	177.3	161.6	198.8
3	147.3	197.6	198.8	161.4	200.7

Table 4: The recovery performance of the various protocols for the **CG** benchmark as f is varied

- In receiver-based logging, the recovering process has a prefix of the information needed during recovery; it can use that prefix to quickly roll-forward and in parallel collect the missing data from the remaining processes.

- In sender-based protocols, messages are stored only in the senders' volatile memory and are lost if the sender fails. Consequently, whenever a recovering process needs to acquire one of these missing messages, it stops rolling forward, waiting for the sender to recover to the point at which the message is regenerated and resent. This *stop-and-go* effect becomes significant with increase in f [25].

The performance of the optimistic protocol is nearly unaffected by increase in f. For the hybrid causal protocol, due to asynchronous logging of messages, the logs on stable storage contain only a prefix of the messages that have to be replayed. A recovering process can use the logged information for replay and in parallel collect the missing data. However, when $f > 1$, the *stop-and-go* effect occurs after the log is replayed and consequently, the cost of recovery marginally increases with f.

6 Related Work

The literature contains several systems that have been designed to provide transparent fault-tolerance using rollback-recovery. They include libckpt [22], Fail-safe PVM [19], MIST [5], Co-check [31], Manetho [9] and libft [13]. As opposed to Egida , however, these systems are designed to support only a specific checkpointing or message-logging protocol.

Egida emphasis on code reusability and extensibility is also present in the recently-introduced OTEC simulator [24]. OTEC's object-oriented design provides some degree of code reusability and can be used to compare the performance of different checkpointing and recovery protocols, but only through simulations. The run-time system for rollback-recovery that is closest to Egida in terms of extensibility is RENEW [21], a system that provides a framework for implementing and evaluating different checkpointing protocols. However, RENEW's architecture does not contain reusable components for faciltating the automatic synthesis of arbitrary rollback-recovery protocols.

Egida 's architecture, which allows to generate protocols from a library of basic building blocks, is similar to the one used in other systems—for instance, in xkernel [14] for networking protocols, in Horus [27] and Cactus [12] for distributed computing and group membership protocols, and in COMERA [35] and Quarterware [28] for communications middleware. However, to our knowledge, Egida is the first application of this approach to rollback recovery protocols.

7 Concluding Remarks

We have presented the design and implementation of *Egida*, an object-oriented toolkit designed to support transparent rollback recovery. Egida exports a simple specification language that can be used to express arbitrary rollback recovery protocols. From this specification, Egida authomatically synthesizes an implementation of the specified protocol by glueing together the appropriate objects from an available library of "building blocks".

We have implemented a prototype of Egida and integrated it with the MPICH implementation of the Message Passing Interface (MPI) standard. Through Egida, MPI applications can be made fault-tolerant without any modifications. Conversely, Egida allows to study the robustness and performance of different rollback-recovery protocols with respect to a large set of demanding applications.

Our early experience with the prototype suggests that Egida is extensible, flexible, and facilitates rapid implementations of rollback-recovery protocols with minimal programming effort. We expect that the design of Egida will evolve as we gain more experience specifying and implementing different rollback recovery protocols.

References

[1] L. Alvisi, B. Hoppe, and K. Marzullo. Nonblocking and Orphan-Free Message Logging Protocols. In *Proceedings of the 23rd Fault-Tolerant Computing Symposium*, pages 145–154, June 1993.

[2] K. Birman and T. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(2):47–76, February 1987.

[3] A. Borg, J. Baumbach, and S. Glazer. A message system supporting fault tolerance. In *Proceedings of the Symposium on Operating Systems Principles*, pages 90–99. ACM SIGOPS, October 1983.

[4] R. Butler and E. Lusk. Monitors, Message, and Clusters: the p4 Parallel Programming System.

[5] J. Casas, D. Clark, P. Galbiati, R. Konuru, S. Otto, R. Prouty, and J. Walpole. MIST: PVM with transparent migration and checkpointing. In *3rd Annual PVM Users' Group Meeting*, Pittsburgh, PA, May 1995.

[6] NASA Ames Research Center. NAS Parallel Benchmarks. http://science.nas.nasa.gov/Software/NPB/, 1997.

[7] O. P. Damani and V. K. Garg. How to Recover Efficiently and Asynchronously when Optimism Fails. In *Proceedings of the 16th International Conference on Distributed Computing Systems*, pages 108–115, 1996.

[8] D. Dolev and D. Malki. The Transis Approach to High Availability Cluster Communication. *Communications of the ACM*, 39(4), April 1996.

[9] E. N. Elnozahy and W. Zwaenepoel. Manetho: Transparent rollback-recovery with low overhead, limited rollback and fast output commit. *IEEE Transactions on Computers*, 41(5):526–531, May 1992.

[10] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996.

[11] William D. Gropp and Ewing Lusk. *User's Guide for* `mpich`*, a Portable Implementation of MPI*. Mathematics and Computer Science Division, Argonne National Laboratory, 1996. ANL-96/6.

[12] M. A. Hiltunen and R. D. Schlichting. A Configurable Membership Service. *IEEE Transactions on Computers*, 47(5):573—586, May 1998.

[13] Y. Huang and C. Kintala. Software Implemented Fault Tolerance: Technologies and Experience. In *Proceedings of the IEEE Fault-Tolerant Computing Symposium*, pages 2–9, 1993.

[14] N. C. Hutchinson and L. L. Peterson. The x-Kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, January 1991.

[15] D. B. Johnson and W. Zwaenepoel. Sender-Based Message Logging. In *Digest of Papers: 17 Annual International Symposium on Fault-Tolerant Computing*, pages 14–19. IEEE Computer Society, June 1987.

[16] D. B. Johnson and W. Zwaenepoel. Recovery in Distributed Systems Using Optimistic Message Logging and Checkpointing. *Journal of Algorithms*, 11:462–491, 1990.

[17] T. Y. Juang and S. Venkatesan. Crash recovery with little overhead. In *Proceedings of the 11th International Conference on Distributed Computing Systems*, pages 454–461. IEEE Computer Society, June 1987.

[18] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

[19] J. Leon, A. L. Ficher, and P. Steenkiste. Fail-safe PVM: A protable package for distributed programming with transparent recovery. Technical Report CMU-CS-93-124, School of Computer Science, Carnegie Mellon University, February 1993.

[20] F. Mattern. Virtual Time and Global States of Distributed Systems. In M. Cosnard et. al., editor, *Parallel and Distributed Algorithms*, pages 215–226. Elsevir Science Publishers B. V., 1989.

[21] N. Neves and W. K. Fuchs. RENEW: A Tool for Fast and Efficient Implementation of Checkpoint Protocols. In *Proceedings of the 28th IEEE Fault-Tolerant Computing Symposium (FTCS)*, Munich, Germany, June 1998.

[22] J. S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt:Transparent checkpointing under Unix. In *Proceedings of the USENIX Technical Conference*, pages 213–224, January 1995.

[23] M. L. Powell and D. L. Presotto. Publishing: A reliable broadcast communication mechanism. In *Proceedings of the Ninth Symposium on Operating System Principles*, pages 100–109. ACM SIGOPS, October 1983.

[24] B. Ramamurthy, S. J. Upadhyaya, and R. K. Iyer. An Object-Oriented Testbed for the Evaluation of Checkpointing and Recovery Systems. In *Proceedings of the 27th IEEE Fault-Tolerant Computing Symposium*, pages 194–203, June 1997.

[25] S. Rao, L. Alvisi, and H. Vin. The Cost of Recovery in Message Logging Protocols. In *Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems*, pages 10–18, West Lafayette, IN, October 1998.

[26] S. Rao, L. Alvisi, and H. Vin. Hybrid Message Logging Protocols For Fast Recovery. In *Digest of FastAbstracts of Fault-Tolerant Computing Symposium (FTCS-28)*, pages 41–42, Munich, Germany, June 1998.

[27] R. Van Renesse, T. Hickey, and K. Birman. Design and Performance of Horus: A Lightweight Group Communications System. Technical Report TR94-1442, Cornell University Computer Science Department, August 1994.

[28] A. Singhai, A. Sane, and R. Campbell. Quarterware for Middleware. In *Proceedings of the International Conference on Distributed Computing and Systems (ICDCS'98)*, pages 192—201, Amsterdam, Holland, May 1998.

[29] A. P. Sistla and J. L. Welch. Efficient Distributed Recovery Using Message Logging. In *Proceedings of the Eighth Symposium on Principles of Distributed Computing*, pages 223–238. ACM SIGACT/SIGOPS, August 1989.

[30] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI: The Complete Reference*. Scientific and Engineering Computation Series. The MIT Press, Cambridge, MA, 1996.

[31] G. Stellner. CoCheck: Checkpointing and process migration for MPI. In *Proceedings of the International Parallel Processing Symposium*, pages 526–531, April 1996.

[32] R. B. Strom and S. Yemeni. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems*, 3(3):204–226, April 1985.

[33] R. E. Strom, D. F. Bacon, and S. A. Yemini. Volatile Logging in n-Fault-Tolerant Distributed Systems. In *Proceedings of the Eighteenth Annual International Symposium on Fault-Tolerant Computing*, pages 44–49, 1988.

[34] S. Venkatesan and T. Y. Juang. Efficient Algorithms for Optimistic Crash Recovery. *Distributed Computing*, 8(2):105–114, June 1994.

[35] Y. M. Wang and W. J. Lee. COMERA: COM Extensible Remoting Architecture. In *Proceedings of the 4th Conference on Object Oriented Technologies and Systems (COOTS)*, April 1998.