Application Fault Tolerance with Armor Middleware

Many current approaches to software-implemented fault tolerance (SIFT) rely on process replication, which is often prohibitively expensive for practical use due to its high performance overhead and cost. The Adaptive Reconfigurable Mobile Objects of Reliability (Armor) middleware architecture offers a scalable low-overhead way to provide high-dependability services to applications. It uses coordinated multithreaded processes to manage redundant resources across interconnected nodes, detect errors in user applications and infrastructural components, and provide failure recovery. The authors describe their experiences and lessons learned in deploying Armor in several diverse fields.

Zbigniew Kalbarczyk, Ravishankar K. Iyer, and Long Wang
University of Illinois, Urbana-Champaign

The widespread availability of relatively low-cost, high-performance commercial-off-the-shelf (COTS) hardware makes software-based fault-tolerance approaches very attractive. Improved processor performance enables developers to delegate fault management to software without sacrificing much in application performance. The growing popularity of distributed and networked systems further accelerates the trend toward software-implemented fault tolerance (SIFT) because networks of nodes naturally provide hardware redundancy, which can be exploited to implement efficient fault tolerance. As a result, researchers have proposed several middleware solutions in recent years to provide high-dependability services to applications.

Most such approaches exploit replication or distributed groups of cooperating processes to provide fault tolerance in networks of unreliable components. Replication can improve system reliability, but its inherently high performance overhead — typically between 200 and 900 percent — and potential nondeterminism (due to operating systems' nondeterministic thread scheduling) in executing different replicas often makes it prohibitively expensive for practical use. In addition, maintaining and managing replicas requires significant hardware (multiple processing units, for example) and software resources (such as an extra communication layer to support distributed groups of cooperating processes).

In endeavoring to alleviate replication's shortcomings, we've explored several alternatives for providing low-overhead, scalable, and robust high-dependability solutions to end-user applications and services. Our Adaptive Reconfigurable Mobile Objects of Reliability (Armor) middleware provides a process architecture and runtime environment for managing redundant resources across interconnected nodes,

 Published by the IEEE Computer Society 1089-7801/05/$20.00 © 2005 IEEE

detecting errors in user applications and infrastructural components, and recovering quickly from failures when they occur.[1] (For more on Armor, see www.crhc.uiuc.edu/depend/.) The Armor architecture doesn't preclude the use of replication, but it employs application-aware detection and recovery mechanisms and algorithms that can be customized and configured to meet applications' needs without employing replication. Furthermore, the Armor runtime environment is, by design, self-checking. We formally specified the Armor architecture and used this formalism to establish criteria for safe reconfiguration when adapting error detection and recovery to a wide range of applications and computing environments.[2]

This article presents our experiences in employing Armor to provide fault tolerance to applications ranging from space-borne scientific programs to commercial databases and software executing on mobile devices. Our experience shows that Armor-based approaches can provide fault tolerance to widely diverse applications. Thorough evaluations of the proposed solutions show that we can achieve high error coverage and low performance overhead with acceptable cost in terms of resources and complexity.

Armor Infrastructure

Armors are multithreaded processes internally structured around objects, called *elements*, which contain their own private data and provide elementary functions or services. Every armor process contains a basic set of elements that provide core functionality, including reliable point-to-point messaging between armors, response to *heartbeat messages* (which indicate a given armor process's "liveness"), and the ability to checkpoint armor state.

Armor processes communicate via message passing: the *microkernel* present in each distributes messages between elements within an armor and between the armors in a system. Every incoming message causes the microkernel to spawn a new thread to process the message, and the execution of each thread invokes one or more elements within the armor process. A thread terminates when the armor finishes processing the message or when it sends an outgoing message in response to the original.

Structurally, messages consist of two primary parts: the *microoperation sequence* and the *payload*. Every message carries a series of microoperations to be executed by armors. The microkernel delivers each microoperation in sequence to elements that have subscribed to that operation. During the initialization within the system, each armor establishes a subscription list, which provides mapping between elements and the microoperations each element can process. The microkernel is responsible for maintaining the list at runtime.

Each message contains a general payload area for storing data. Elements can read from and write to the payload fields while processing the microoperations in a message. Thus, elements can exchange information with one another within a single execution thread. This information exchange doesn't interfere with other execution threads because each thread manipulates its own payload.

While processing a microoperation, an element can update its local state or the state of the payload fields; it can also change an armor process's control flow by adding new microoperations to the current sequence. This modular, event-driven architecture permits developers to customize an armor process's functionality and fault-tolerance services (detection and recovery) according to the application's needs.

Figure 1a illustrates an example of the basic Armor configuration:

- The *fault-tolerance manager* (FTM) initializes an Armor-based environment's working configuration, maintains registration information on all armor objects and application processes, and initiates recovery from armor and node failures.
- The *heartbeat armor* runs on a node that is separate from the FTM. It detects failures in the FTM by periodically polling for liveness, and then initiates FTM recovery.
- A *daemon armor* runs on each node in the network, serving as a gateway for armor-to-armor communication and detecting runtime failures of local armors (those running on the same node as the daemon).
- The *execution armor* launches local application processes, detects their failures, and performs recovery.

Several armor processes constitute the self-checking runtime environment, and each plays a specific role in the detection-and-recovery hierarchy offered to the system and the application.

Application Support

Processes constructed around the Armor architecture are organized to form a distributed SIFT environment for protecting user applications in the network. The armor processes provide error-

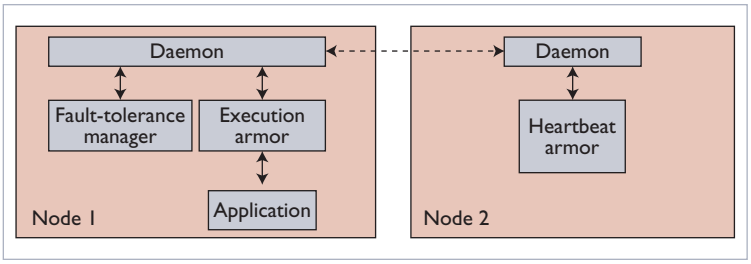

Figure 1. Example Armor configuration. The basic Armor runtime environment scales to two nodes. Each node runs a daemon armor, which serves as the communication gate for armor-to-armor communication. The fault-tolerance manager (FTM) coordinates all actions within the armor environment. It is monitored by the heartbeat armor, which initiates recovery in case of FTM failure. The execution armor watches an application's progress and triggers recovery if it fails.

detection and recovery services through techniques that depend on the degree to which the application is integrated with the SIFT runtime environment.

Two qualities characterize a given fault-tolerance technique:

- where it resides with respect to the application — whether internally (application checkpointing, for example) or externally (monitoring the application process for abnormal termination, for example), and
- how transparent it is to the application.

Although generic external solutions that are transparent to the application can provide significant protection, tighter integration with an application usually facilitates more sophisticated error detection and recovery.

Armor offers three levels of application support.

- *Level 1: Transparent and external support.* Widely applicable error-detection and recovery techniques don't require developers to modify applications. Example capabilities include detecting application-process failures (crashes) and restarting failed application processes, whether on the same node or another.
- *Level 2: Transparent extension of standard libraries.* The Armor infrastructure hardens standard libraries (such as operating system calls or C library calls) with additional capabilities. Example capabilities include detecting application hangs, automatic recovery of broken TCP/IP socket connections, and IP address fail-over.
- *Level 3: Instrumentation with Armor APIs.* The Armor infrastructure defines an API through which applications interact with armor processes. Developers can tightly integrate fault-tolerance mechanisms with the application processes, permitting a greater degree of customization than is available at the other two levels. Example internal mechanisms include embedded armors, memory-state checkpoints, and application-specific self-tests (for instance, control-flow checking).

Level-2 and level-3 techniques often require developers to establish communication channels between the application and the Armor runtime environment. The user application initializes the communication channel through an explicit call to an Armor-level API (for level-3 support) or through an enhanced function call from an existing library (for level-2 support). For example, an extended version of `MPI_Init()` can establish the communication channel in addition to initializing runtime support for message-passing interface (MPI) support.

On the Armor side, a special element called `AppIpcMgmt` connects to the communication channel and sends and receives messages through it. The communication channel can be unidirectional (in which case, it supports only application-to-armor communication) or bidirectional (as required for application checkpointing and recovery, for example).

Case Studies

From its inception in 1997 through several years of research, the Armor architecture evolved into a mature software middleware. Since then, we've deployed Armor-based fault-tolerance solutions for multiple applications and computing environments. Our experiences with these various projects have informed and guided subsequent deployments as we've refined the Armor architecture. Although the following examples focus on detection and recovery services exposed to the application, note that the Armor infrastructure is self-checking — that is, it also transparently handles failures of armor processes.

SIFT for Space-Borne Applications

The Remote Exploration and Experimentation (REE) project at NASA's Jet Propulsion Lab (JPL) intends to use a cluster of COTS processors to provide high-performance computing on spacecraft. The REE cluster executes scientific distributed applications under the protection of an Armor-based SIFT environment. Key requirements for the

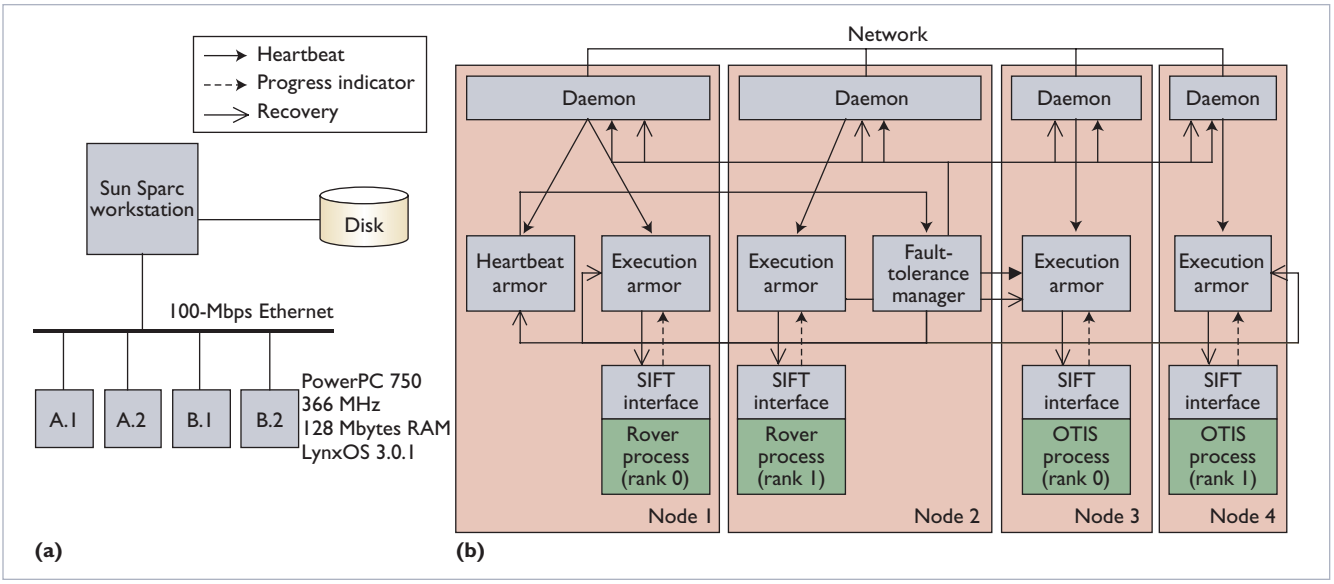

Figure 2. Software-implemented fault-tolerance (SIFT) architecture for executing MPI applications. (a) The Remote Exploration and Experimentation (REE) test bed is a network with four nodes, each running LynxOS on a PowerPC 750. The processors communicate with each other via Ethernet, and a remote Sun workstation serves as the repository for program executables and data. (b) This SIFT architecture supports two MPI applications (from NASA's Mars Rover and Orbiting Thermal Imaging Spectrometer [OTIS] missions) executing on the test bed. Arrows depict the relationships among the application and armor processes. For example, the application sends progress indicators to the execution armor, and the FTM sends heartbeats to the daemon armor process.

environment include providing detection and recovery from crash and hang failures in the applications and SIFT processes, and minimizing error propagation within nodes and across the network.

Figure 2a depicts the REE project's four-node experimental test bed, which comprises two boards (A and B), each with two PowerPC 750 processors running the Lynx real-time operating system. The processors communicate with each other via Ethernet, and a remote file system on a Sun workstation stores program executables and application input and output data. Figure 2b shows the SIFT architecture for executing two MPI applications on the test bed: the Mars Rover texture-analysis program (Rover) and the Orbiting Thermal Imaging Spectrometer (OTIS), which extracts land temperature from thermal images taken by sensors.

The REE SIFT environment primarily leverages level-1 techniques to protect the applications: the execution armor detects crashes by intercepting operating system signals (within a supervising armor process) on abnormal application-process termination and coordinates recovery across the multiple MPI application processes.

Execution armors also include elements that detect application hangs. To enable hang detection, each application process establishes a one-way communication channel (level-3 application source-code instrumentation) with the execution armor. Applications use the SIFT interface (shown in Figure 2b) to periodically update progress indicators, which the execution armor polls at fixed intervals. If the progress indicator value stagnates, the execution armor initiates application recovery.

We conducted extensive fault-injection experiments to assess error-detection coverage as well as recovery and performance overhead due to the armor processes. Our key findings indicate that

- the Armor-based SIFT environment adds negligible overhead (less than 2 percent) to applications' execution time during failure-free runs;
- Armor successfully recovers from all *correlated failures* of multiple processes (taking into account the correlated failures, the mean overhead on application-execution time increases to 5 percent); and
- assertions within the armor processes (verifying an armor process ID's validity, for example), coupled with incremental checkpointing, reduce the number of system failures due to data-error propagation by up to 42 percent.

Further details on this work are available elsewhere.[3]

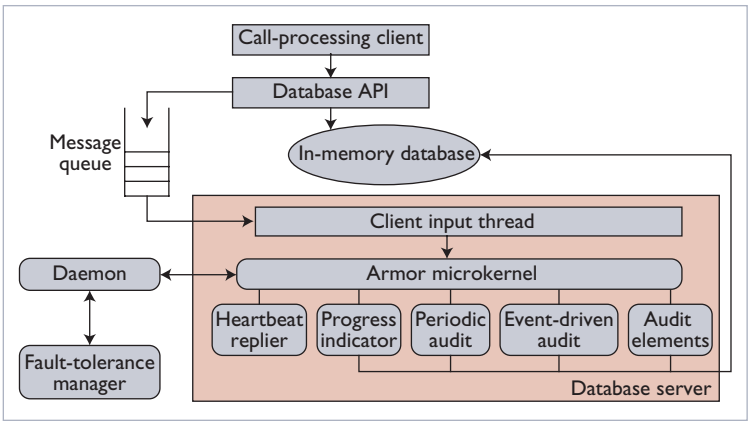

Figure 3. Armor-based database audit framework. A call-processing application interacts with the database and database server under the supervision of the armor processes that provide continuous data auditing for consistency and integrity.

Wireless Telephone Network Controller

The wireless telephone network controller is an example of a telecommunication system that employs data audits to improve service availability to end users. The system integrates a call-processing application to handle activities associated with each voice or data connection and an associated database that maintains the application's configuration parameters and resource-usage status.

We worked with a network controller vendor to apply an extended Armor-based environment. We used elements to implement a database-auditing framework, which applies level-1 monitoring of application processes, including managers, to control the data audits and determine appropriate recovery actions.

The framework supports both periodic and triggered (those performed in response to specified events) data audits. We incorporated the auditing functionality into the call-processing application through a combination of level-2 techniques (extending a client-side database API to invoke data audits before allowing a write request to the database, for example) and level-3 techniques (such as augmenting a database server to enable response to armor messages).

Figure 3 illustrates the overall design of the database audit process and its interaction with other system components. The elements include the heartbeat, progress indicator, audit elements (which encapsulate a set of error-detection and recovery techniques), periodic audit, and event-driven audit.

We extended the database server process to include the reconfigurable Armor architecture, through which developers can add special data-auditing elements. The database server's configuration is called an *embedded armor* (level-3 extensions) because the core Armor architecture is integrated into an existing application process. From the SIFT environment's perspective, an embedded armor behaves like any other armor process: it sends and receives armor messages. Armor-derived functionality (for example, the database server can respond via the *heartbeat replier* element to heartbeat messages from the daemon armor) executes separately from the application's threads; thus, the application doesn't need to be aware of the embedded armor's extra processing capabilities.

Employing level-2 extensions to the client-side database API, we added an interprocess communication channel (message queue) between the database API and the database server (augmented with the embedded armor) to transmit events from client activities (write requests, for example) to the data-auditing elements in the database server's embedded armor. A dedicated client-input thread (part of the database server) acts as the interface to the other components by translating information from external entities, such as the database client, into armor messages. The embedded armor serves as the vehicle to provide audit functionality and includes elements that implement specific audit-triggering, error-detection, and recovery techniques.

We implemented an emulated call-processing client to measure both the armor audit's effectiveness in protecting database clients from errors and the performance impact of doing so. We conducted the experiments on a Sun UltraSparc-2 system, inserting random bit errors into the database at various rates.

Our results show that database audits are useful in removing data errors and preventing error propagation; moreover, the audits don't break down, even when the error rate is high. In our experiments, data audits achieved 85 percent error coverage and reduced error propagation from the database to call-processing clients by a factor of 5 (from 63 percent to 13 percent). However, the average call setup time in the client process increased by 69 percent — from 160 to 270 milliseconds — due to the processing time required by the audits. More details on this study are available elsewhere.[4]

Main Memory Database System

Commercial main-memory databases (MMDBs) are designed to support the development of high-performance, fault-resilient applications that require

Related Work in Reliable Distributed Systems

Researchers have done significant work in the area of providing fault tolerance in distributed and networked systems. Wensley's software-implemented fault-tolerance (SIFT) approach was one of the earliest attempts to do so substantially through software rather than hardware.[1] In SIFT, copies of a program execute independently on redundant loosely synchronized processing modules. At the beginning of each iteration, the consuming task performs a majority vote on its inputs — the previous iteration's outputs — before proceeding.

Delta-4 was one of the earliest efforts to build a dependable, open, distributed system using off-the-shelf components.[2] Armor leverages this early experience while providing customizable and scalable software middleware.

Approaches that use *process groups* have attracted considerable attention for process replication. The field of group communication has subsequently emerged to address the issues of how to manage memberships within groups and how groups communicate with each other. Although group communication protocols such as the Isis toolkit[3] and Ensemble[4] can achieve reliability, fault tolerance is essentially a side effect of the replication approach. In contrast, Armor is designed specifically to provide fault-tolerance services to applications.

Cactus offered a framework for constructing dependability protocols (communication and voting) from a set of primitive microprotocols connected through an event-driven communication infrastructure.[5] In Cactus, the notion of reconfigurability doesn't extend beyond communication protocols. The Armor architecture, on the other hand, employs a reconfigurable design for all its functionality, giving the developer an opportunity to incorporate fault-tolerance techniques in system components other than the communication infrastructure (for example, checkpointing application state).

Eternal[6] and Aqua[7] employed the Common Object Request Broker Architecture standard to support process replication. Corba addresses dependability issues through extensions that support replications and group communication. Armor doesn't preclude replication, but it attempts to provide fault tolerance through a wide range of error-detection and recovery mechanisms available to both applications and armor processes.

References

1. J. Wensley, "SIFT Software Implemented Fault Tolerance," *Proc. Fall Joint Computer Conf.*, Am. Federation of Information Processing Societies (AFIPS) Press, vol. 41, 1972, pp. 243–253.
2. D. Powell et al., "The Delta-4 Approach to Dependability in Open Distributed Computing Systems," *Proc. 18th Ann. Int'l Symp. Fault-Tolerant Computing*, IEEE CS Press, 1988, pp. 245–251.
3. K. Birman and R. van Renesse, *Reliable Distributed Computing with the Isis Toolkit*, IEEE CS Press, 1994.
4. M. Hayden, "The Ensemble System," PhD dissertation, Computer Science Dept., Cornell Univ., 1998.
5. J. He et al., "Providing QoS Customization in Distributed Object Systems," *Proc. IFIP/ACM Int'l Conf. Distributed Systems Platforms*, ACM Press, 2001, pp. 351–372.
6. L. Moser, P. Melliar-Smith, and P. Narasimhan, "A Fault Tolerance Framework for CORBA," *Proc. 29th Ann. Int'l Symp. Fault-Tolerant Computing* (FTCS 29), IEEE CS Press, 1999, pp. 150–157.
7. M. Cukier et al., "AQuA: An Adaptive Architecture that Provides Dependable Distributed Objects," *Proc. 17th Symp. Reliable and Distributed Systems* (SRDS 17), IEEE CS Press, 1998, pp. 245–253.

concurrent access to shared data. In addition to user data, the database maintains the control structures (which we call SystemDB) – the file tables necessary for correct operation, for example. The system employs multiple locks to guarantee mutual exclusion when user processes access control structures. When a client or service crashes while holding a lock to shared data, the database can remain in an inconsistent state. To bring the system back into a consistent state, the cleanup process initiates a major recovery and restarts the database. This can take up to tens of seconds, depending on the data file sizes, and can significantly degrade system availability.

To reduce or eliminate the cases in which major recovery is needed, we apply the Armor-based checkpointing framework to enable efficient monitoring and recovery of SystemDB. The SystemDB checkpointing employs a combination of level-2 (modifying the database APIs to enable the collection and restoration of database state) and level-3 (embedded armor) mechanisms.

We tested the system using two different checkpointing algorithms: *incremental* and *delta*. With incremental checkpointing, a dedicated armor process collects the control structures' post-transaction state changes and merges them with the current checkpoint. At recovery time, the armor uses the checkpoint directly to restore the correct state. With delta checkpointing, the armor process preserves pretransaction state (before any updates occur) of the control structures accessed by a given transaction, and uses it as a current (delta) checkpoint. At recovery time, the armor merges the current state of control structures in the shared memory with the delta checkpoint (stored within the armor) to restore the SystemDB state.

The test bed consists of a Sun Blade 100 workstation running the Solaris 8 operating system on

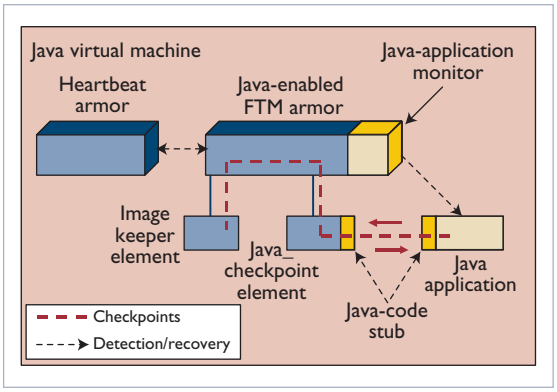

Figure 4. Java-enabled Armor. The java_checkpoint element takes checkpoints and sends them to the image keeper via armor communication channels, enabled by a custom Java-code stub in the java_checkpoint element and Java application. The image keeper element maintains application checkpoints. The heartbeat armor detects and recovers from failures in Java-enabled fault-tolerance manager (FTM) armors.

a 500-MHz UltraSparc-II CPU with 128 Mbytes of memory. Our results indicate that the armor processes and runtime environment can provide low-overhead checkpointing (less than 2 percent for a typical workload, in which more than 50 percent of the operations are read-only) and recovery of control structures in a commercial MMDB. Moreover, our proposed solution virtually eliminates cases that require major recovery and significantly improves availability. Indeed, we estimated (based on the measured database recovery time) database availability to be 99.999 percent — under the assumption that one error required major recovery per week. Further details are available elsewhere.[5]

New Challenges and Applications

We're currently using Armor in several ongoing projects that expand the infrastructure's applicability to new platforms and applications, including mobile devices and large embedded-network systems. The techniques applied in these projects demonstrate the strength of a customizable architecture in addressing emerging challenges for achieving fault tolerance.

Fault-Tolerance Services for Embedded Java

Java's strong typing makes it an excellent choice for developing robust software — particularly for mobile and embedded devices. We've customized Armor to provide error-detection and recovery services to embedded applications. We use embedded armors to support monitoring, crash detection, restarting, and checkpointing of Java applications. A key challenge has been to adapt our process-oriented architecture to work with Java threads.

Figure 4 illustrates the Java-enabled Armor configuration. This simplified version of the generic configuration reduces performance overhead and reduces footprint size. We achieved this customization relatively easily by incorporating only those elements the application requires (analysis of application, system software, and hardware specifics allows us to determine which armor elements an application needs) and collocating them in a single armor (the FTM in Figure 4).

We developed an additional element, `java_checkpoint`, to take application checkpoints and send them to the *image keeper* (which stores application checkpoints) via armor communication channels (enabled by a custom Java-code stub, which we embed in the `java_checkpoint` element and Java applications). The checkpointing interface the applications use consists of only two function calls: `updateState()` and `restoreState()`. These let the developer specify what data to checkpoint, when to checkpoint, and when to recover in case of application failure.

Our initial test bed implementation runs on a Sharp Zaurus SL-5600 PDA, with a 400-MHz ARM processor and 32 Mbytes of RAM. The PDA executes a Linux kernel with Sun's Java 2 Micro Edition HotSpot Java virtual machine. We installed a Java-based Web browser and a simple game on the test platform, and used the Armor infrastructure to checkpoint their state. The Web browser, for example, calls the checkpointing element to update state every time the URL changes.

Large-Scale Real-Time Embedded Systems

We've deployed Armor middleware to provide a test platform for a trigger and data-acquisition system for the BTeV accelerator-based high-energy physics experiment (http://www-btev.fnal.gov/public/hep/detector/rtes/), which studies matter–antimatter asymmetries in the decays of particles containing the bottom quark. The BTeV system must be available 24 hours a day, 7 days a week, and although the system can tolerate losing a small fraction of the data for a short time period, significant data loss is unacceptable.

Our initial focus has been on integrating the BTeV event dispatch and processing subsystem with the Armor infrastructure to provide fault-

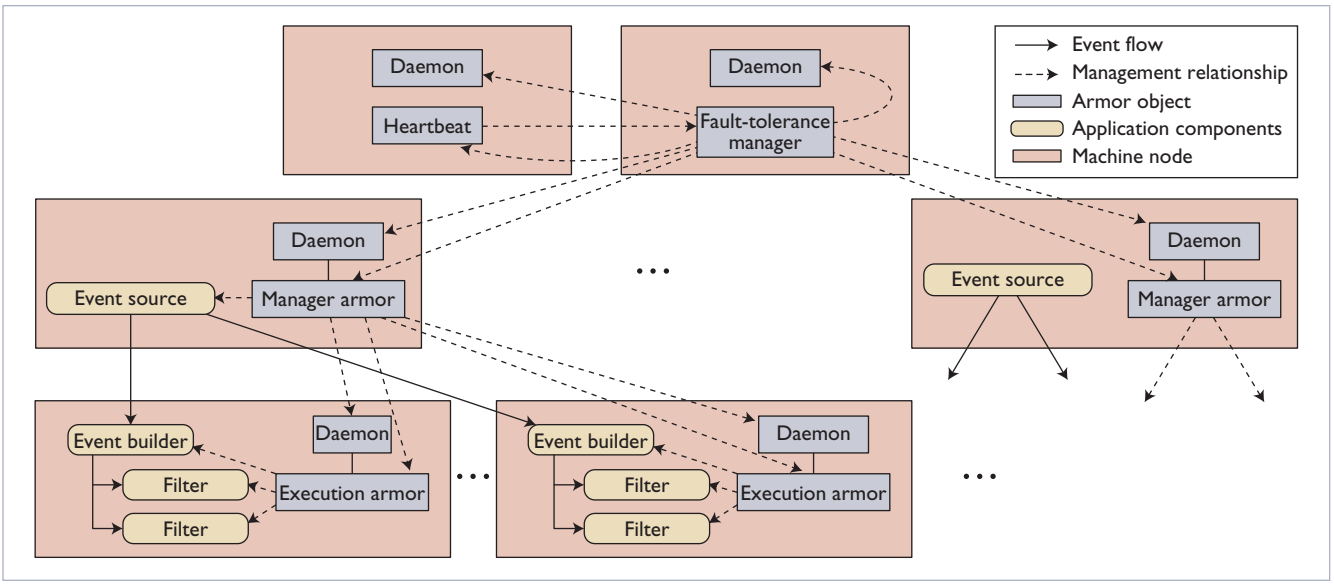

Figure 5. The BTeV event dispatch and processing subsystem with Armor-provided fault tolerance. Hierarchically organized armor processes can provide detection and recovery to large-scale embedded systems. The solid arrows indicate event flows, and the dashed arrows indicate the management relationships. For example, a dashed arrow from a manager armor to an execution armor means that the manager armor initiates recovery from the execution armor's failures.

tolerant operations (see Figure 5). The subsystem collects event data in real time via numerous simple processing entities (roughly 3,000 in the final system) and delivers it to a Linux farm running approximately 2,500 desktop or workstation machines (in the final system). A dedicated process, called an *event source*, acts as an input gateway on each node and accepts the data for further distribution and processing. Each event source dispatches events to multiple *event builders*, which then deliver them to *event filters* (applications) for processing.

The yellow boxes in Figure 5 are application components, and the gray boxes are Armor objects. The solid arrows indicate the event flows, and the dashed arrows indicate management relationships. In a large network, the developer establishes the Armor management hierarchy by outfitting each *manager armor* with management elements that let us establish the management hierarchy (with the FTM on top) and assign each manager a distinct subset of computing nodes in the system to supervise. Although the figure shows only three hierarchy levels, we can extend the structure to scale and configure fault-tolerance services as needed for specific applications.

Discussion

Table 1 (next page) summarizes the Armor applications and fault-tolerance techniques we've implemented (including some examples we didn't cover in this article due to space limitations). The armor processes' reconfigurability and the Armor runtime environment's self-checking functionality are the key factors that let us provide solutions for multiple application domains.

With regard to the first factor, all of the example applications show that level-1 techniques are widely useful, regardless of application specifics and requirements. We implement these generic fault-tolerance techniques as elements that reside in armor processes that are external (and, hence, transparent) to the applications.

The applicability of level-2 and level-3 techniques depends on a given application's characteristics and the degree to which it is integrated with the armor. For example, an embedded armor solution works well as a framework for database auditing and checkpointing because we can get full access to client-side database APIs without changes to the application code.

Still, implementing progress indicators or heartbeats requires few additions to the code. A level-2 implementation wraps standard library function calls to augment the progress-indicator functionality (for example, the application might send a progress indicator message whenever it called the `write()` function on a socket). This approach maintains transparency at the source-code level, although it requires relinking if the

Table 1. Applications of Armor middleware.

Application	Description and platform	Fault-tolerance techniques														
		Level 1			Level 2				Level 3							
		Crash detection (OS-based)	Process restart (same node)	Process migration (fail-over)	TCP connection recovery	IP address fail-over	Data audits (periodic event-driven)	Exception detection/ interception	Embedded armor	Heartbeat replier	Process state checkpointing	Database checkpointing	Progress indicator (hang detection)	Control flow signature	Full-fledged Armor
Armor processes	Armor runtime environment	✓	✓	✓				✓		✓	✓			✓	✓
MPI-based space-borne scientific applications	SIFT for protecting MPI applications (LynxOS on PowerPC)	✓	✓	✓									✓		
Wireless telephone network controller	Data-auditing framework for protecting database server (Solaris on Sparc)	✓	✓	✓			✓		✓	✓			✓		
Main memory database system (MMDB)	Checkpointing of the control structures in MMDB (Solaris on Sparc)	✓	✓	✓					✓	✓		✓			
Real-time embedded systems	Large-scale system for data acquisition and processing (Linux on Pentium)	✓	✓	✓					✓	✓	✓		✓		
Java-based Web browser and a game	J2ME applications on a handheld device (Linux on Pentium)	✓	✓	✓				✓	✓		✓				
Microsoft PowerPoint	Checkpointing of MS applications (Windows on Pentium)	✓	✓	✓				✓			✓				
Streaming audio	Client-server applications on wireless/wireline networks (Linux on Pentium)	✓	✓	✓	✓					✓					
Dynamic Host Configuration Protocol server	Full-fledged Armor implementation of DHCP (Solaris on Sparc)	✓	✓	✓					✓	✓	✓		✓	✓	✓
Telecommunication middleware	Application fail-over with preserving IP address (Linux on PowerPC)	✓	✓	✓		✓			✓	✓	✓				

augmented libraries are statically linked to an application's executable file.

In contrast, a level-3 implementation requires the developer to alter application source code — adding explicit calls to a progress-indicator API, for example. Similarly, we could deploy a full Armor-based implementation of a Dynamic Host Configuration Protocol server because the server partitions and performs operations on the application data in a relatively straightforward manner, allowing us to create an element set to implement DHCP services.

With regard to user applications, the Armor architecture lets us create application-specific fault-tolerance mechanisms and encapsulate them as elements. We can insert these elements into an existing armor process following well-established procedures, thus enabling the user to tailor fault-tolerance services according to the application's needs without detailed knowledge of Armor internals.

With regard to the second factor, Armor's self-

checking property, we have shown that the middleware on which fault-tolerance techniques are based must itself be fault tolerant to limit or prevent error propagation. To demonstrate, we recently conducted an error-injection-based study on the Ensemble group communication system (www.cs.cornell.edu/Info/Projects/Ensemble/). Application developers use Ensemble to provide a robust communication layer to support replication. Our study shows that approximately 5 percent of errors propagate across the network, and roughly 1.5 percent of these cause the remote nodes that receive the erroneous packets to crash.[6] Armor processes can virtually eliminate such scenarios. Moreover, Armor middleware can successfully recover from correlated failures (or multiple failures occurring in short succession).

The lessons we've learned in developing Armor extend beyond this architecture and can inform the design of any SIFT services. Case studies employing real-world applications demonstrate that any robust middleware for implementing such services must follow several key design principles. At a minimum, these include

- *reconfigurability* (static and dynamic) to facilitate customization to suit the application and available resources;
- *self-checking and recovery* to transparently handle errors within the middleware, eliminate (or, at least, minimize) error propagation, and recover from correlated errors;
- *hierarchical fault and error management* to scale with a large number of nodes or components; and
- *decoupling of fault management from the application* to ensure low overhead.

Our goal is to build on this base and extend the Armor middleware to address new domains. The software is continuously evolving, and our current work includes applying the technology to cellular devices and security technologies.

Acknowledgments

The authors thank their colleagues, past and present, who contributed to the research described in this article. They also thank their past and current research sponsors: the Jet Propulsion Laboratory (contract 961345); the US National Science Foundation (CCR 00-86096 ITR, CCR 99-02026, ACI 0121658 ITR/AP, and ACI CNS-0406351); the US Office of Naval Research and the Defense Advanced Research Projects Agency (MURI Grant N00014-01-1-0576); Gigascale Systems Research Center (GSRC/MARCO); and the Motorola Corporation. Finally, they thank Tamara O'Neill for her editorial assistance.

References

1. Z. Kalbarczyk et al., "Chameleon: A Software Infrastructure for Adaptive Fault Tolerance," *IEEE Trans. Parallel and Distributed Systems*, vol. 10, no. 6, 1999, pp. 560–579.
2. K. Whisnant, Z. Kalbarczyk, and R. Iyer, "A System Model for Reconfigurable Software," *IBM Systems J.*, vol. 42, no. 1, 2003, pp. 45–59.
3. K. Whisnant et al., "The Effects of an Armor-Based SIFT Environment on the Performance and Dependability of User Applications," *IEEE Trans. Software Eng.*, vol. 30, no. 4, 2004, pp. 257–277.
4. S. Bagchi et al., "A Framework for Database Audit and Control Flow Checking for a Wireless Telephone Network Controller," *Proc. Int'l Conf. Dependable Systems and Networks*, IEEE CS Press, 2001, pp. 225–234.
5. L. Wang, Z. Kalbarczyk, and R.K. Iyer, "Checkpointing of Control Structures in Main Memory Database Systems," *Proc. Int'l Conf. Dependable Systems and Networks* (DSN 04), IEEE CS Press, 2004, pp. 687–692.
6. C. Basile et al., "Group Communication Protocols under Errors," *Proc. 22nd Int'l Symp. Reliable Distributed Systems* (SRDS 03), IEEE CS Press, 2003, pp. 35–46.

Zbigniew Kalbarczyk is principal research scientist at the University of Illinois, Urbana-Champaign. His research interests include automated design, implementation, and evaluation of dependable and secure computing systems. Kalbarczyk received a PhD in computer science from the Bulgarian Academy of Sciences. He is a member of the IEEE and the IEEE Computer Society. Contact him at kalbar@crhc.uiuc.edu.

Ravishankar K. Iyer is director of the Coordinated Science Laboratory and George and Ann Fisher Distinguished Professor of Engineering at the University of Illinois, Urbana-Champaign. His research interests include reliable and secure computing, measurement and evaluation, and automated design. Iyer received a PhD from the University of Queensland, Australia. He is an IEEE fellow, an associate fellow of the American Institute for Aeronautics and Astronautics, and a member of the ACM, Sigma Xi, and the IFIP Technical Committee on Fault-Tolerant Computing (WG 10.4). Contact him at iyer@crhc.uiuc.edu.

Long Wang is a PhD student in the Department of Electrical and Computer Engineering at the University of Illinois, Urbana-Champaign. His research interests include fault tolerance, system reliability, and system analysis. Wang received an MS in computer science from the University of Illinois, Urbana-Champaign. Contact him at longwang@crhc.uiuc.edu.