Julep: an Environment for the Evaluation of Distributed Process Recovery Protocols

Lawrence R. Klos Golden G. Richard III
{lklos, golden}@cs.uno.edu
Department of Computer Science
University of New Orleans
New Orleans, LA 70148

Abstract

Julep is an object-oriented testbed designed for implementation and analysis of process recovery protocols. It is written in Java, and runs as a layer underneath a Java-based distributed application. Only minor modifications to a typical distributed application are necessary to use Julep as a communication mechanism. Julep is designed to allow new process recovery mechanisms to be quickly incorporated, permitting accurate comparison between mechanisms for specific distributed applications on specific hardware platforms. A novel aspect of Julep is its UDP-based object communication service, which implements "unbreakable" communication channels. Julep can be used as a testbed to compare the performance of particular recovery mechanisms, as a framework within which new recovery mechanisms can be implemented and tested, or as an infrastructure to make existing distributed applications fault tolerant. In its most basic form, Julep can be used as a reliable object-based communication service.

1. Introduction

Software fault-tolerance protocols based on process recovery have been under development for over twenty years. Research in this area has matured to the point where developers are often required to offer guarantees of reliability for recovery of failed processes in distributed applications, and rightly so. With current knowledge, throwing out the interim processing of a distributed application because a few processes have failed and must be restarted is simply wasteful.

Many software mechanisms have been developed that can be combined in various ways to implement a wide variety of these protocols. Julep was designed to study temporal diversity protocols that rely on a fail-stop node model. Individual mechanisms developed for temporal diversity process recovery, such as logical and vector clocks, various types of message logging and checkpointing, as well as the process recovery protocols that can be created from these mechanisms are fully described in [5] and [6]. Each protocol that has been developed offers its own set of guarantees regarding the process failure scenarios it can handle. Each protocol also incurs two different sets of overhead costs: one for standard operation, and one for operation during the recovery of a process.

Rigorous analysis and comparison of process recovery protocols in terms of their overhead costs in specific settings is slowed by the fact that the set of mechanisms required to implement a particular protocol usually must be added to already complex software systems on a custom basis. Developers are seldom willing to implement two protocols for the sake of comparison.

The need to quickly implement and evaluate or compare various software process recovery protocols was the impetus behind the creation of the Julep distributed process recovery environment. Julep is fully implemented in Java. Java provides several benefits to the Julep system. The multithreading necessary for simultaneous message sending and reception is easily implemented, simple mechanisms are available for shared memory, synchronization and sockets, and a standard implementation will work across heterogeneous networks.

Julep allows specific comparisons between process recovery protocols in terms of the efficiency of the protocols and the amount of system overhead the protocols require. The overhead for a particular protocol can depend, in part, on the physical characteristics of the hardware the application is running on. Because Julep is written in Java and runs on any hardware with an installed JVM, a protocol's performance can be analyzed on a particular distributed

hardware configuration, and the results used to optimize the protocol for that configuration.

Julep provides two main services to a distributed application. First, it provides a reliable object-based communication service, supplying each application node with a set of message send/receive primitives. All system communication passes through, and is manipulated by, the Julep layer. The second service Julep provides is that of system node management. This service dynamically tracks liveness, status, and location for every node in the distributed application, allowing the system to quickly adapt to node failures and changing node locations. At the application level, nodes are identified without reference to location, allowing Julep to always route messages to a migrating node's current location. Julep implements a configurable blocking message send functionality. If a node receiving a message fails midway through the message reception and restarts on a remote machine, the communication channel is dynamically rerouted to its new location and the send event will eventually successfully complete. This is the basis for Julep's "unbreakable" communication channels.

These services are combined in Julep to create an environment within which process recovery protocols can be easily implemented and compared. Most process recovery protocols rely on storing and manipulating system messages in some manner; Julep's provision of a message transport service allows it to manipulate all system communication to satisfy this requirement of any process recovery protocol.

The organization of this paper is as follows: Section 2 gives an overview of the Julep system, while Sections 3 and 4 describe Julep's Manager and Communication services in more detail. Section 5 describes two process recovery protocols that have been implemented into the Julep system with Section 6 giving a performance comparison between them. Section 7 describes related work, and finally, Section 8 gives conclusions and future work.

2. Overview of Julep

Julep has three main components; a central Manager, JulepEndpoints, and UDPObjectTransfer components. The central Manager component provides a system-wide node location lookup service for all nodes, the JulepEndpoint component provides a communication endpoint, and the UDPObjectTransfer component provides low-level object send/receive functionality. Figure 1 shows a modified UML diagram of all the component classes in the Julep system. For a complete discussion of all of Julep's components and their functionality, see [11].

Each communication endpoint must have a system wide unique identifier. Distributed applications instantiate JulepEndpoint objects, which provide an application level interface to the Julep system. The unique identifier is supplied to the JulepEndpoint constructor. Each JulepEndpoint discovers its current location and then registers with the central Manager, where its ID and location are bound and made available for lookup by other nodes in the system. Upon startup, the JulepEndpoint also initiates a HeartBeatSender thread, whose purpose is to periodically send "I'm alive" heartbeats to the Manager. This allows the Manager to passively track the state of the node. The JulepEndpoint object can also actively change its state with a terminate( ) method call to the Manager at any time, which will remove it from the Julep system.

The primary service provided by a JulepEndpoint object is reliable send/receive message passing primitives. A node uses these primitives for all network communication. When a registered node wants to send a message to a peer node, it calls its JulepEndpoint's high level send() method, passing it the ID of the receiver node's JulepEndpoint and an object. If the JulepEndpoint has previously sent a message to the specified destination, it will find the destination's physical location in its cache. If this is the first message sent to a particular destination, the JulepEndpoint will request the receiver's location from the Manager, and cache the reply. This caching reduces communication overhead and the load on the Manager.

Associated with each JulepEndpoint object is a UDPObjectTransfer object. This object takes care of all low-level communication tasks for the node. The JulepEndpoint object passes a message to be transmitted, along with the receiver's location, to its UDPObjectTransfer object, which handles the details of transforming the object to be transferred into a series of packets. For message reception, a communication endpoint's UDPObjectTransfer object runs a daemon thread that awaits incoming message packets. When it receives all packets that constitute an incoming message, it transforms them back into the original object, and invokes its DeliverDriver to deliver the object to a message queue in the associated JulepEndpoint object. The DeliverDriver provides a location for specific message-logging implementations to be incorporated. The receiver invokes JulepEndpoint's receive() primitive to access the message from this queue.

The Manager component is a standalone object. Its purpose is to register and track the status of all nodes in the system, and act as a server for node location requests. If an individual node fails,

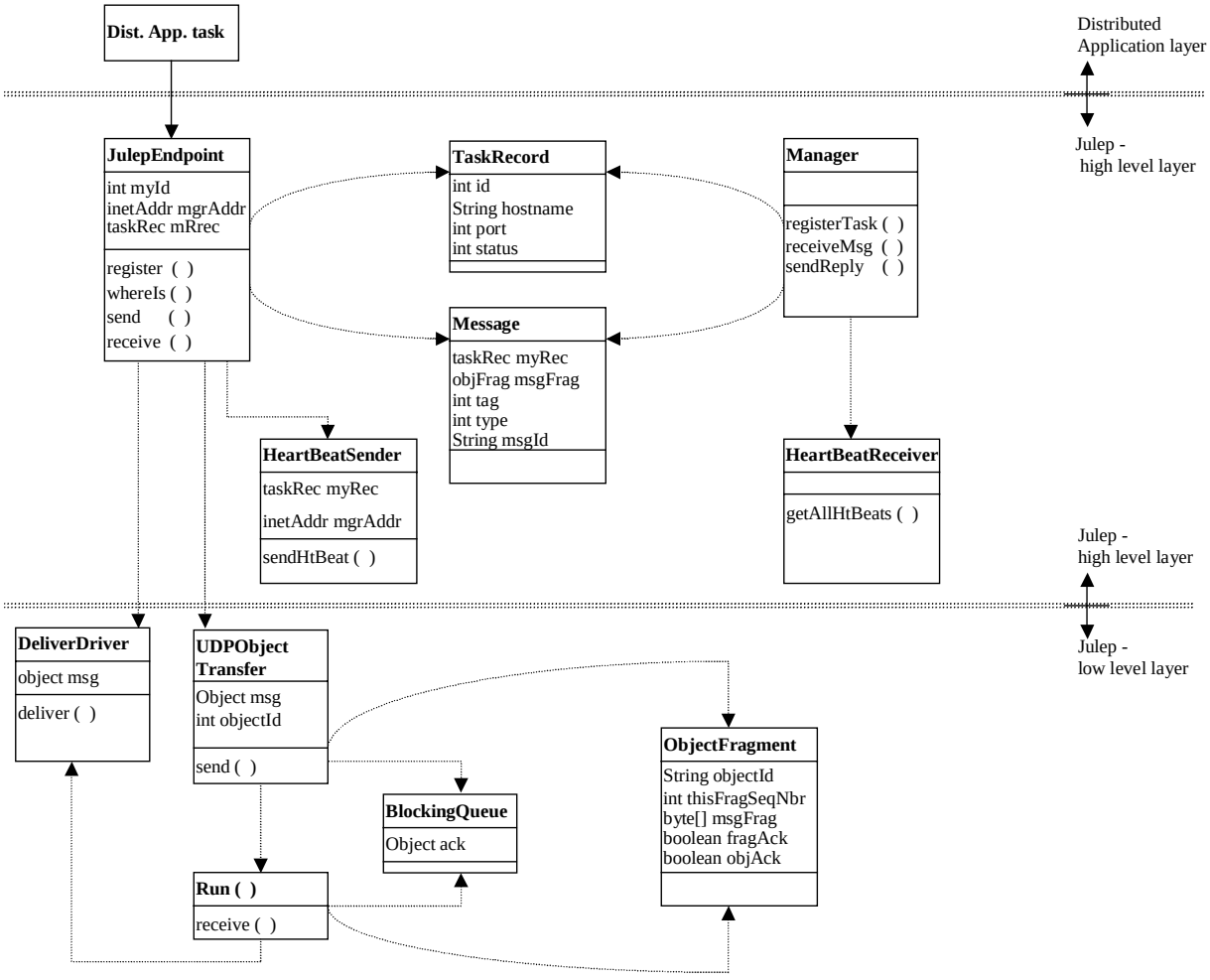

Figure1. The core Julep classes.

the Manager will eventually react to the failure and flag the node as DEAD. Nodes attempting to communicate with the failed node will block. Eventually, when the failed node re-registers (possibly from a new location), the Manager will respond to a location request with the (new) location of the destination and the sender will unblock.

3. Julep's Manager Service

The central Manager component in Julep provides a node location lookup service using node ID/location binding. The Manager tracks liveness of all registered nodes in the system passively through periodic heartbeat reception, and actively through registration and termination commands. The Manager is fault-tolerant: on receiving node update messages the Manager logs the changes to stable storage. In the event of Manager failure, a restarting Manager will access stable storage and restore its state prior to failure.

The Manager can have additional system-wide services programmed into it. For example, it currently has the ability to function as a Group Membership Service Provider with a comprehensive set of group management commands. The JulepEndpoint component provides primitives for group communication that work in conjunction with the Manager to guarantee delivery and guarantee ordering constraints. Also, the JulepEndpoint component can invoke the standard send() method with a group ID instead of a node ID; this will send a single unicast message to the least loaded member node in the group. A visualization system has also been developed for Julep. The group communication and visualization facilities of Julep are not discussed further in this paper due to space constraints.

4. Julep's Communication Service

Julep is very flexible in the services it can provide to a distributed application. The fault-tolerant

mechanisms included in the system can simply be turned off, or used selectively to augment normal operation. A prototype utilizing Julep for distributed observation of personnel movements is described in [12]. In this prototype, GPS location data reflecting personnel movements in a field of operations is multicast to remote viewers, where it appears in context through a 3D virtual reality interface. Julep was used as the network communication service, and its message logging mechanism was enabled at the remote viewing stations for purposes of recording rather than fault-tolerance. The remote stations can "replay" logged events at any future time, for purposes of analysis and training.

For performance reasons, Julep uses the connectionless UDP protocol on top of IP for all message passing. The UDP transport service has several drawbacks when compared to TCP, another common transport service. UDP packet delivery service is unreliable, and UDP packets have a fixed maximum length. However, it has advantages over TCP in that no overhead is required for the establishment of a communication channel and no file handles are consumed to support connections. For a node that has failed and is restarting, this can be a significant savings in overhead, especially if the node was communicating with many peer nodes at the time of its failure. The low level message transport layer implemented in Julep is essentially a software layer on top of UDP that handles fragmenting Java objects into packets, and ordering, sending and reassembly of these packets at the receiver. Resends of lost packets are automatically handled, and duplication of packets or out of order packet reception is handled transparently at the receiver side. Time limits for packet reception and message reconstruction are configurable, so slow hosts or networks can be accommodated. This protocol layer in Julep makes the UDP protocol reliable and able to process messages of unlimited size.

To send a message to another node, an application node will first invoke its JulepEndpoint object's send() method, which in turn, invokes its UDPObjectTransfer object's send() method. When the UDPObjectTransfer send() method is invoked, the message object is first transformed into a byte array. This byte array is fragmented into a series of UDP packet sized fragments, which are loaded, along with message ID and fragment sequencing information, into a series of UDP packets. The packets are then sent over the network to the daemon thread associated with the receiver node's UDPObjectTransfer object. This daemon thread replies with an *ack* UDP packet for every received packet. The sending side overlaps sending (or resending if necessary), of the message fragment packets with receipt of the corresponding *ack* packets. Once the last packet has been sent and its corresponding *ack* packet received, the sending side blocks waiting for an object *ack* packet. At this stage, the receiving side sequences the byte array fragments, transforms the byte array back into the original message object and makes it available for reception. Once initial delivery of the message has completed, an object *ack* packet is sent to finally unblock the sending process.

If a configurable timeout limit is reached while waiting for *acks*, control on the sender side passes back to the send() method which re-attempts the message send. This process is repeated a configurable number of times. After the limit on send attempts is reached, control returns to the sender's JulepEndpoint object, which clears its cache of the receivers location, sends a new location request to the Manager, and re-attempts the message send. This is the "unbreakable" message channel functionality; if the receiver node fails in mid-reception and restarts on a different machine, upon registration with the Manager the sender's message send operation will discover the receiver's new location and complete the transmission.

5. Currently Implemented Recovery Protocols

Recovery protocols generally require process checkpointing in some form. Currently, checkpointing in Julep can be handled by a set of application specific fault-tolerant design patterns at the level of the distributed application, as described in [17]. Many researchers are working on mechanisms to add persistence to Java objects at the level of the JVM as well. We are investigating the incorporation of such a mechanism into Julep, but are currently concerned about portability issues.

The first protocol that has been implemented in Julep is the Pessimistic Receiver Based Logging (PRBL) Protocol [8]. Under this protocol, the node receiving a message is the one responsible for logging it to stable storage before processing it. This protocol requires the Julep system to prevent a message sender from unblocking until the receiver has successfully stored the message into a message log. If a node fails and restarts, it simply recreates its state prior to failure by replaying messages from the message log on disk. The main advantage of this protocol is that node recovery has a smaller system-wide impact, since there is no required rollback or interaction with peer nodes: the recovering node recreates its state with only local information. The main disadvantage is that pessimistic logging to disk during normal operation is a very time consuming task. Worse, the

logging occurs on the critical path of a node's processing; the sender and receiver are blocked until logging is complete.

The Pessimistic Sender Based Logging (PSBL) Protocol [10] has also been implemented in the Julep System. This protocol requires the node sending the message to log it to volatile storage before sending it. The receiver node transmits back to the sender the value of a Received Sequence Number (RSN) counter, and increments the counter for the next incoming message. In the event of a node failure, upon restart the failed node sends "recovery" messages to all system nodes. Each node will check its queue of sent messages in volatile storage, pulling and resending any previously sent to the recovering node, along with their individual RSN numbers. This allows the recovering node to order the incoming messages into the original order of reception, to recreate the same state it had prior to failure. The main advantage of this protocol is that during normal operation message logging to volatile storage is very fast. The disadvantages are that only one node at a time may fail, and that the recovery protocol for a failed node is more complex; a failed node must interact with all nodes from which it previously received messages. Thus overall system-wide impact of a recovering node is greater under this protocol.

6. Protocol Performance Comparison

Experimental evaluation of Julep and the various recovery protocols we have implemented is a work-in-progress. We present some preliminary results in this section.

Protocol comparison is usually specific to a particular distributed application. The overhead of a protocol is a combination of the amount of additional communication required, the size of the messages, and the complexity of the algorithms for both normal and recovery operations. The first two factors can vary to a great degree between different applications. If a distributed application relies on a pattern of synchronous communication between nodes, the entire system could be forced to halt until a failed node completes its recovery operation. In order to establish as neutral an environment as possible, an application was selected in which the failure of a task would not affect the speed of processing to a greater extent than simply having one less node available for work, but where the recovery of the failed node was essential to the completion of the overall task.

The initial test application consisted of two client nodes requesting tasks, and one server node dispensing them. The tasks were simulated, with a configurable processing "wait" time, to allow comparison between processor intensive tasks and network communication intensive tasks. The distributed application protocol required each client to send an initial "request" message to retrieve a task from the server. The server responds with a "task" message containing the task. The client completes the task, sending a "done" message to the server, and receiving an "ack" message in response. The client is then free to request another task. The task processing "wait" time was configured to 50 ms for the following experiment.

A System Run time represents the span of time from the point after the first client "request" message reaches the server and the point after the twentieth "done" message reaches the server. For the Recovery System Run, one client is controlled by a SuperClient that starts the client, kills it after it completes four tasks (i.e., sends 8 messages and receives 8 messages), waits 500 milliseconds, then restarts it. Recovery for this client would then consist of retrieving and processing the eight previously received messages and then rejoining the active system. All system messages were limited to the size of one UDP packet. The single message time shown is the total one-way latency for a message transfer.

The protocol comparison test was performed on a set of Sun Ultra 30 workstations with 296 MHz processors, connected by a 100Mb/s Ethernet network under lightly loaded conditions. Each application task ran on a separate machine, with the Julep Manager also running on its own machine.

Test Results:

Normal System runs with no node failures: The values presented below were averaged from a series of runs:

System Run with no logging occurring: 3511 ms.

- Avg. time required to send one message: 24 ms.

PSBL Protocol: 3548 ms.

- Avg. time required to send one message: 23 ms.

PRBL Protocol: 7250 ms.

- Avg. time required to send one message: 172 ms.

It is clear that with no node failures, the overhead for the PSBL protocol is negligible. On the other hand, overhead for the implementation of the PRBL protocol is large and system message passing overhead is significantly higher as well. This is expected, given the overhead of disk access to perform logging.

System Runs with one node failure and recovery: For node recovery comparisons, the recovery time differential between the two logging protocols consisted of the time each protocol requires to retrieve all previously received messages. Once the messages are retrieved, the two protocols process

them in the same manner. In both protocols, the recovery of messages occurs on the second instantiation of the node's JulepEndpoint object, in its constructor. This time is listed below each protocol. The main protocol time listed is the measured span of time between the point after the first client "request" message reaches the server and the point after the twentieth "done" message reaches the server, with one node failing and restarting after completing four tasks. This node then contributes to task processing after restarting.

PSBL Protocol:	5708 ms.
- time for 2nd instantiation of JulepEndpoint object:	1930 ms.
PRBL Protocol:	7861 ms.
- time for 2nd instantiation of JulepEndpoint object:	640 ms.

For the PRBL Protocol, message retrieval for a recovering node consists of reading the previously logged messages from disk. The time the PRBL protocol requires for the second JulepEndpoint instantiation is very close to the time required for the original instantiation, once the extra time for initial class loading is accounted for. The instantiation/message retrieval time for the PSBL protocol is significantly greater than that of the PRBL protocol, which makes sense, given the relative complexity of the PSBL recovery/message retrieval algorithm, and the fact that the recovering node must block for a synchronous reply to its "recovery" messages sent to the live nodes. Not included in the time listed for the PSBL Protocol's second instantiation of JulepEndpoint is the time the live nodes in the system must take to handle the "recovery" message from the restarting node, though it is included as part of the overall PSBL system run with a process failure and recovery.

The total System Run with one node failure and recovery for the PRBL protocol represents a 611 ms. increase over the total PRBL System Run with no node failures, while the same run for the PSBL protocol shows a 2160 ms. increase. The relatively large amount of time required by the PSBL protocol for message retrieval by the single recovering node would account for most of this time, with the rest taken by the required participation of other live nodes in the recovery of the failed node.

In this initial test, the amount of overhead the recovery mode of the PSBL protocol requires is greater than that of the PRBL protocol, however the normal processing differential between the two is so large that, for this case, the PSBL protocol still takes less time for the overall system run. Increasing the number of clients would most likely have a drastic effect on the differential in recovery times, however. Future tests will quantify this.

A central problem in optimizing the recovery operations of the PSBL protocol lies in minimizing the blocking that occurs in live system nodes as they process a recovery message from a restarting node. For this implementation, the live nodes start a separate thread to handle the recovery message, so the live node processing is minimally impacted. Julep has a node specific "verbose" feature that allows close monitoring of the progress and the interactions of all individual threads making up a node, as well as any interactions of that node with other nodes in the distributed application. This feature is a great benefit in determining if implemented protocol optimizations are performing in their intended manner under specific recovery scenarios.

Optimization of Julep's communication service is an immediate goal. After this is accomplished, a series of benchmark tests will be performed to determine the overhead of Juleps communications layer, comparing its efficiency to that of TCP.

7. Related Work

A research area related to Julep's focus is the construction of simulated environments to analyze distributed process recovery protocol performance under various fault models. One such project is OTEC [14], an object oriented testbed, which utilizes the DEPEND [7] system simulation tool with fault injection capability. The distributed application to be tested with the OTEC system could be a version simulated from user specifications, or the actual implemented application. OTEC provides a set of basic process recovery components layered on top of DEPEND that can be composed in various ways to implement hybrid fault-tolerant protocols. OTEC is similar to the Julep system, in this compositional approach to protocols.

A centralized approach to testbed environments for implemented real time distributed protocols is explored in the Cesium [1] project, which simulates a distributed environment by executing all tasks in a single address space. The centralization of processing allows greater control over execution of tasks, monitoring and analysis.

Another research area closer in design to Julep is the insertion of a distributed software layer in an existing operational environment, with the purpose of analyzing a targeted protocol layer. ORCHESTRA [4], is a real time distributed protocol testing environment with fault injection capabilities for testing of real time protocol guarantees. ORCHESTRA operates as a software layer inserted below a targeted protocol layer in a distributed system. This approach

allows ORCHESTRA to manipulate and filter all messages passing between the targeted software layer on a node and the distributed system at large. Julep is designed using a similar layered approach to take advantage of message access and manipulation at each node.

Ideally, these simulated and implemented environments will allow fault-tolerant characteristics of protocols and applications to be efficiently evaluated during the prototype or the operational phase. Workload generation, fault injection and data analysis components in simulated environments will theoretically provide a laboratory-like experimental setting allowing for much greater precision in specific condition testing than would be possible in an operational environment. For example, deterministically repeatable simulation of concurrent faults in separate components of an application are feasable through environment simulation. Such a level of specificity is necessary for fault-tolerant protocols for real time systems, because by their nature they must offer specific detailed ironclad guarantees. A testbed environment for such protocols must have the ability to model specific faults and analyze results for unique systemwide scenarios. Test environments and fault injectors for such environments often must be platform, or even application specific, because of the specificity of protocol guarantees undergoing analysis. Development time for such test environments is correspondingly large. Julep was not designed for such systems. Julep was meant for general fault-tolerant protocol overhead testing in a fail stop environment. Initial goals set during Julep's design were that the testbed have maximum portability, run in a true operational environment, and that fault-tolerant protocols be quickly and easily composed from scratch, given the less stringent requirements of non-real time protocols.

ReSoFT[19] by SOHAR Inc. is a system augmentation environment for fault-tolerance analysis. It provides an object oriented library of basic reusable fault-tolerance components defined by object oriented analysis, and implemented in Ada95. Components supporting the software fault-tolerance categories of design diversity and data diversity have been implemented. ReSoFT also provides a set of graphical tools to give system building, monitoring, and fault injection capability. A central goal of ReSoFT is to allow users to take existing distributed software systems into the environment, add fault-tolerant components and evaluate the resulting system.

Julep, in contrast, is specifically designed to study temporal diversity fault-tolerance strategies. Julep's focus is the comparison and evaluation of specific fault-tolerant mechanisms and protocols within this domain under the widest range of specific hardware platforms possible, enabling protocol evaluation for a specific application and platform. For this reason Julep is written in Java, and can run on a variety of platforms. The analysis capability Julep provides is that of protocol overhead analysis, rather than fault injection oriented protocol error handling analysis.

In the past decade several testbeds have been implemented to evaluate SWFT mechanisms for various strategies [2][13]. These testbeds studied strategies other than temporal diversity however. Some of these testbeds were non-portably implemented, limiting their applicability.

A more recent object oriented fault-tolerance framework was developed in [20]. It was designed to be an off-the-shelf component which can be added to an existing system, rather than a testbed for new components. The framework is configurable by the user to provide a range of fault-tolerance levels. A single fault-tolerance strategy is employed in the framework; Specialized N-modular redundancy (SNMR), a design diversity strategy. The system utilizes a base model hierarchically modified for increasingly sophisticated fault-tolerance protocols at proportionate system overhead costs. It can handle fail-stop, general and Byzantine failure modes, and various system replication models.

8. Conclusions and Future Work

In this paper we have described Julep, a Java-based testbed for implementation and evaluation of distributed process recovery protocols. Julep provides "unbreakable" communication channels, which automatically re-route messages when communication endpoints fail and recover. Julep can also be used as an infrastructure for developing fault-tolerant distributed applications, or for adding fault-tolerance to existing Java applications. With no recovery algorithms activated, the Julep system can function strictly as a reliable object-based communication facility.

While implementation of representative process recovery protocols in Julep is a work-in-progress, we have presented some initial results for two simple protocols in this paper. More sophisticated protocols [e.g., 15] are under construction.

We are currently evaluating and optimizing Julep's communication protocol in an attempt to reduce communication latency. Julep is also being extended to support wireless environments. This work includes adding support for location-based groups and changing the single central manager to a dynamic group [3] of managers. The latter poses some interesting design decisions, because we must address intra-manager communication, the expansion and contrac-

tion of the manager group to handle queries and heartbeats, and node registration transfer in the case of manager failures.

Source code for Julep is available by request. Please contact the authors for more information or keep an eye on http://www.cs.uno.edu/~golden.

References

[1] G. Alvarez and F. Cristian, Centralized Failure Injection for Distributed, Fault-tolerant Protocol Testing. In Proc. of the 17th IEEE International Conference on Distributed Computing Systems (ICDCS '97), Baltimore, Maryland, May 1997.

[2] A. Avizienis et al., The UCLA DEDIX System: a Distributed Testbed for Multiple-Version Software. In 15th International Symposium on Fault-Tolerant Computing (FTCS 15) , pages 126-134, June, 1985.

[3]Kenneth P. Birman, Building Secure and Reliable Network Applications. Manning Publications Co., 1996.

[4] S. Dawson, F. Jahanian, and T. Mitton, ORCHESTRA: A Fault Injection Environment for Distributed Systems. University of Michigan Technical Report CSE-TR-318-96, EECS Department.

[5] G. Deconinck, J. Vounckx, R. Cuyvers and R. Lauwereins. Survey of Checkpointing and Rollback Techniques. Technical Report O.3.1.8 and O.3.1.12, ESAT-ACCA Laboratory, Katholieke Universiteit Leuven, Belgium, June 1993.

[6] E. N. Elnozahy, D. B. Johnson and Y. M. Wang. A Survey of Rollback-Recovery Protocols in Message Passing Systems. Technical Report CMU-CS-96-181, Department of Computer Science, Carnegie Mellon University, 1996.

[7] K. K. Goswami, R. K. Iyer and L. Young, DEPEND: A Simulation-Based Environment for System Level Dependability Analysis. In IEEE Transactions on Computers, Vol. 46, no. 1, pages 60-74, January, 1997.

[8] Y. Huang, and C. Kintala, A Software Fault-tolerance Platform. In Practical Reusable Software, Ed. B. Krishnamurthy, pages 223-245. John Wiley & Sons, 1995.

[9] P. Jalote. Fault-tolerance in Distributed Systems. Prentice Hall, 1994.

[10] D.B. Johnson and W. Zwaenepoel. Sender-based message logging. In Proc. IEEE Fault-Tolerant Computing Symp., pages 14-19, 1987.

[11] L. Klos, G. G. Richard III, Z. Xu, Julep: A Framework for Reliable Distributed Computing in Java. University of New Orleans Technical Report UNOCS-TR99-01.

[12] R. Ladner, M. Abdelguerfi, G. G. Richard, III, L. Klos, B. Liu, K. Shaw, A Distributed Virtual Reality Prototype for Real Time GPS Data. Accepted to Second International Symposium on TeleGeoProcessing, Nice, France, May, 2000.

[13] J. M. Purtilo and P. Jalote, An Environment for Developing Fault-Tolerant Software. In IEEE Transactions on Software Engineering, Vol. 17, No. 2, pages 153-159, February, 1991.

[14] B. Ramamurthy, S. J. Upadhyaya, and R. K. Iyer, An Object-Oriented Testbed for the Evaluation of Checkpointing and Recovery Systems. In 27th International Symposium on Fault-Tolerant Computing (FTCS 27), pages 194-203, June, 1997.

[15] G. G. Richard III and M. Singhal, Complete Process Recovery: Using Vector Time to Handle Multiple Failures in Distributed Systems. In IEEE Concurrency – Parallel, Distributed & Mobile Computing, pages 50-59, April-June 1997.

[16] G. G. Richard III, Efficient Vector Time with Dynamic Process Creation and Termination. In Journal of Parallel and Distributed Computing, v55, no. 1, pages 109-120, Nov. 1998.

[17] G. G. Richard III and S. Tu. On Patterns for Practical Fault-tolerant Software in Java. In 17th IEEE Symposium on Reliable Distributed Systems, West Lafayette, Indiana October 20-23, 1998.

[18] J. Rumbaugh, I. Jacobson and G. Booch, The Unified Modeling Language Reference Manual, Addison-Wesley, 1998.

[19] K. S. Tso and E. H. Shokri, ReSoFT: a Reusable Software Fault-tolerance Testbed. In Pacific Rim International Symposium of Fault-Tolerant Systems, Newport Beach, CA, pages 98-103, December, 1995.

[20] I-L. Yen, I. Ahmed, R. Jagannath, and S. Kundu, Implementation of a Customizable Fault-Tolerance Framework. In The First IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, Kyoto, Japan, April, 1998.