Recovery Management in QuickSilver

Roger Haskin, Yoni Malachi[1], Wayne Sawdon, and Gregory Chan[2]
IBM Almaden Research Center
San Jose, California 95120-6099

The last several years has seen the emergence of two trends in operating system design: ***extensibility***: the ability to support new functions and machine configurations without changes to the kernel, and ***distribution***: partitioning computation and data across multiple computers. The QuickSilver distributed system, being developed at the IBM Almaden Research Center, is an example of such an extensible, distributed system. It is intended to provide a computing environment for various people and projects in our lab, and to serve as a vehicle for research in operating systems and distributed processing.

One price of extensibility and distribution, as implemented in QuickSilver, is a more complicated set of failure modes, and the consequent necessity of dealing with them. In traditional operating systems, services (*e.g.*, file, display) are intrinsic pieces of the kernel. Process state is maintained in kernel tables, and the kernel contains explicit cleanup code (*e.g.*, to close files, reclaim memory, and get rid of process images after hardware or software failures). QuickSilver, however, is structured according to the client-server model, and as in many systems of its type, system services are implemented by user-level processes that maintain a substantial amount of client process state. Examples of this state are the open files, screen windows, address space, etc., belonging to a process. Failure resilience in such an environment requires that clients and servers be aware of problems involving each other. Examples of the way one would like the system to behave include having files closed and windows removed from the screen when a client terminates, and having clients see bad return codes (rather than hanging) when a file server crashes. This motivates a number of design goals:

- Properly written programs (especially servers) should be resilient to external process and machine failures, and should be able to recover all resources associated with failed entities.
- Server processes should contain their own recovery code. The kernel should not make any distinction between system service processes and normal application processes.
- To avoid the proliferation of ad-hoc recovery mechanisms, there should be a uniform system-wide architecture for recovery management.
- A client may invoke several independent servers to perform a set of logically related activities (a ***unit of work***) that must execute ***atomically*** in the presence of failures, that is, either all the related activities should occur or none of them should. The recovery mechanism should support this.

In QuickSilver, recovery is based on the database notion of ***atomic transactions***, which are made available as a system service to be used by other, higher-level servers. This allows meeting all the above design goals. Software portability is important in the QuickSilver environment, dictating that transaction-based recovery be accessible to conventional programming languages rather than a special-purpose one such as Argus [Liskov84]. To accommodate servers with diverse recovery demands, the low-level primitives of commit coordination and log recovery are exposed directly rather than building recovery on top of a stable-storage mechanism such as in CPR [Attanasio87] or recoverable objects such as those in Camelot [Spector87] or Clouds [Allchin&McKendry83].

The QuickSilver recovery manager is implemented as a server process, and contains three primary components:

- **Transaction Manager**- a component that manages commit coordination by communicating with servers at its own node and with transaction managers at other nodes.
- **Log Manager**- a component that serves as a common recovery log both for the Transaction Manager's commit log and server's recovery data.
- **Deadlock Detector**- a component that detects and resolves global deadlocks and resolves them by aborting offending transactions.

Of these three components, the Transaction Manager and Log Manager have been implemented and are in use. The Deadlock Detector, based on a design described by Obermarck [Obermarck 82], has not been implemented, and is mentioned here only to show where it fits into our architecture.

The basic idea behind recovery management in QuickSilver is as follows: Clients and servers interact using a message-passing interprocess communication (IPC) facility. Every IPC message belongs to a uniquely identified transaction, and is tagged with its transaction ID (*Tid*). Servers tag the state they maintain on behalf of a transaction with its *Tid*. IPC keeps track of all servers receiving messages belonging to a transaction, so the Transaction Manager (TM) can include them in the commit protocol. TM's commit protocol is driven by calls from the client and servers, and by

---

1 Author's present address is Ready Systems, Inc., Palo Alto, CA.

2 Author's present address is MIT Laboratory for Computer Science, Cambridge MA

failure notifications from the kernel. Servers use the commit protocol messages as a signalling mechanism to inform them of failures, and as a synchronization mechanism for achieving atomicity. Recoverable servers call the Log Manager (LM) to store their recovery data and to recover their state after crashes.

A painful fact is that transactions as they are normally thought of are a rather heavyweight mechanism. Using transactions as a single, system-wide recovery paradigm depends upon being able to efficiently accommodate simple servers. For example, the window manager, virtual terminal service, and address space manager (loader), have ***volatile*** internal state that does not need to survive system crashes. These servers only require a simple signalling mechanism to inform them of client termination and failures. Often, such servers have stringent performance demands. If telling the loader to clean up an address space is expensive, command scripts will execute slowly. The recovery manager has several important properties that help it address its conflicting goals of generality and efficiency.

- The recovery manager concentrates recovery functions in one place, eliminating duplicated or ad-hoc recovery code in each server.
- Recovery management primitives (commit coordination, log recovery, deadlock detection) are made available directly, and servers can use them independently according to their needs.
- The transaction manager allows servers to select among several variants of the commit protocol (one-phase, two-phase). Simple servers can use a lightweight variant of the protocol, while recoverable servers can use full two-phase commit.
- Servers communicate with the recovery manager at their node. Recovery managers communicate among themselves over the network to perform distributed commit. This reduces the number of commit protocol network messages. Furthermore, the distributed commit protocol is optimized (*e.g.*, when all servers at a node are one-phase or read only) to minimize log forces and network messages.
- The commit protocols support mutual dependencies among groups of servers involved in a transaction, and allows asynchronous operation of the servers.
- The log manager maintains a common log, and records are written sequentially. Log I/O is minimized, because servers can depend on TM's commit record to force their log records.
- A block-level log interface is provided for servers that generate large amounts of log traffic, minimizing the overhead of writing log records.
- Log recovery is driven by the server, not by LM. This allows servers to implement whatever recovery policy they want, and simplifies porting servers with existing log recovery techniques to the system.

QuickSilver is installed and running in daily production use on 47 IBM RT-PC's in the computer science department at IBM Almaden Research Center. In addition to the QuickSilver group, other research projects in several IBM labs are using QuickSilver as an environment to develop applications and network-based services.

The recovery manager has been implemented and is being used as the recovery mechanism for all QuickSilver servers. Experience with the system has confirmed that recovery management overhead is negligible and not perceptible to users, and that the mechanism is efficient enough to be used for servers with very stringent performance demands.

Our intent is to pursue development of the QuickSilver recovery manager in the following areas:

- **Deadlock Detection.** As we mentioned earlier, the Deadlock Detection component of the Recovery Manager has not been implemented. We anticipate beginning this work shortly.
- **High-Performance Servers.** The "block access" log interface reduces the number of calls to the log manager, but causes sparser utilization of log blocks and more log block writes. Considerably more performance analysis is necessary to evaluate the benefit of block access for servers like the file system that potentially log large amounts of data.
- **Nested Transactions.** QuickSilver presently does not include a nested transaction mechanism. The utility of a mechanism such as that proposed by Moss [Moss85] is clear, and we intend to investigate implementing one.
- **Recoverable Object Managers.** We recognize the merit of systems like Camelot and Argus that make it easy to define and use recoverable objects. It is relatively straightforward to implement recoverable object managers on top of the QuickSilver recovery primitives. We intend to explore a language-directed facility for defining and using recoverable objects, perhaps in the context of a language like C++.

Bibliography

[Allchin & McKendry 83] Allchin, J. E., McKendry, M. S., Synchronization and recovery of actions, *Proceedings of the Second ACM Symposium on Principles of Distributed Computing* (August 1983) pp. 31-44.

[Attanasio 87] Attanasio, C. R., CPR supervisor support for relational database facility, IBM Technical Report RC 12416 (January 1987).

[Liskov 84] Liskov, B., Overview of the Argus language and system, MIT Laboratory for Computer Science (February 1984).

[Spector 87] Spector, A., et. al., Camelot: A distributed transaction facility for Mach and the internet- An Interim Report, CMU Technical Report CMU-CS-87-129 (June, 1987).

[Moss 85] Moss E. B., *Nested Transactions: an Approach to Reliable Distributed Computing,* MIT Press (1985).

[Obermarck 82] Obermarck R., Distributed deadlock detection algorithm, *ACM Transactions on Database Systems* **Volume 7, Number 2** (June 1982) pp. 187-208.