CMU-CS-86-167

Avalon: Language Support for Reliable Distributed Systems

Maurice P. Herlihy and Jeannette M. Wing
Department of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890
3 December 1986

Abstract

Avalon is a set of linguistic constructs designed to give programmers explicit control over transaction-based processing of atomic objects for fault-tolerant applications. These constructs are to be implemented as extensions to familiar programming languages such as C++, Common Lisp, and Ada; they are tailored for each base language so the syntax and spirit of each language are maintained.

This paper presents an overview of the novel aspects of Avalon/C++: (1) support for testing transaction serialization orders at run-time, and (2) user-defined, but system-invoked, transaction commit and abort operations for atomic data objects. These capabilities provide programmers with the flexibility to exploit the semantics of applications to enhance efficiency, concurrency, and fault-tolerance.

Maurice P. Herlihy
(412) 268-2584
Herlihy@c.cs.cmu.edu.arpa

Jeannette M. Wing
(412) 268-3068
Wing@c.cs.cmu.edu.arpa

Word count: 4428 (3945 in text, 484 in display).

Conference area: (b) specification, design, testing, verification of reliable software

1. Introduction

Large networks of computers supporting both local and distributed processing are emerging as the computing environments of choice. Application programs running in these environments concurrently access shared, distributed, and possibly replicated data. Examples of such applications include airline reservations, electronic banking, process control, and campus-wide networks of workstations. Such applications must be designed to cope with failures and concurrency, ensuring that the data they manage remain *consistent*, that is, are neither lost nor corrupted, and *available*, that is, accessible even in the presence of failures such as node crashes and network partitions.

A widely-accepted technique for preserving consistency in the presence of failures and concurrency is to organize computations as sequential processes called *transactions*. Transactions are *atomic*, that is, serializable and recoverable. *Serializability* means that transactions appear to execute in a serial order, and *recoverability* means that a transaction either succeeds completely or has no effect. A transaction that completes all its changes successfully *commits*; otherwise it *aborts*, and any changes it has made are undone.

Although transactions are widely used in the database community, demonstrating that they can be a foundation for general purpose distributed systems remains a challenge and is currently of active interest. Appropriate programming language support for application programmers would greatly enhance the usability and thus, generality, of such systems.

Avalon is a set of linguistic constructs designed as extensions to familiar high-level programming languages such as C++ [Stroustrup 86], Common Lisp [Steele 84], and Ada [Dod 83]. The extensions are tailored for each base language, so the syntax and spirit of each language are maintained. The constructs include new encapsulation and abstraction mechanisms, as well as support for concurrency and recovery. The decision to extend existing languages rather than to invent a new language was based on pragmatic considerations. We felt we could focus more effectively on the new and interesting issues of reliability and concurrency if we did not have to redesign or reimplement basic language features, and we felt that building on top of widely-used and widely-available languages would facilitate the use of Avalon outside our own research group.

This paper presents an overview of some of the more novel aspects of Avalon/C++. In Section 2, we survey the main features of Avalon, and in Section 3 we focus on Avalon's two principal innovations: (1) support for *hybrid atomicity* [Weihl 84], a concurrency control mechanism in which transactions are serialized in the order they commit, and (2) user-defined, but system-invoked, transaction commit and abort operations for atomic data objects. In Section 4, we give two examples showing how these constructs can be combined to enhance concurrency, and in Section 5 we close by comparing the Avalon constructs with other distributed programming languages and systems. We are currently implementing the Avalon/C++ constructs on top of Camelot [Spector et al. 86], a distributed transaction management system being built at Carnegie Mellon University.

2. Overview of Avalon

A program in Avalon consists of a set of **servers**, which resemble Argus *guardians* [Liskov 83]. A server encapsulates a set of objects and exports a set of **operations** and a set of **constructors**. A server resides at a single physical node, but each node may be home to multiple servers. An application

program explicitly creates a server at a specified node by calling one of its constructors. Rather than sharing data directly, servers communicate by calling one another's operations. An operation call is a remote procedure call with call-by-value transmission of arguments and results. When a server receives an operation call, it creates a short-lived "light-weight" process to execute the operation. A server can also provide a special **background** operation called by the system after it is created, as well as a special **recover** operation called by the system when the server recovers from a crash. Objects may be stable or volatile. *Stable* objects survive crashes, while *volatile* objects do not. Syntactically, a server resembles a C++ class definition, where the objects correspond to class members, the operations correspond to member operations, and the constructors correspond to constructors. At the statement level, Avalon provides primitives to begin and end transactions, either in sequence or in parallel. Each transaction is identified with a process.

Avalon also supports nested transactions. A transaction commits only if all its children commit or abort; a transaction that aborts aborts all its children, even those that have committed. A transaction's effects become permanent only when it commits at the top level. Thus, a subtransaction's effects need not be written to stable storage until its top-level transaction commits. Nested transactions can be used to make applications more robust. For example, if a subtransaction aborts, the parent transaction need not abort, but can execute an alternative subtransaction. Nested transactions also increase the level of concurrency within a single transaction since subtransactions may execute concurrently.

We use standard tree terminology when discussing nested transactions: parent, child, ancestor, descendant, etc. A transaction is considered to be its own ancestor, but not its own proper ancestor, and similarly for descendants. For purposes of discussion, all transactions are considered to be descendants of a distinguished *root* transaction that never commits or aborts. A *top-level* transaction is a child of the root transaction. If B is an ancestor of A, then A has *committed with respect to* B if every transaction that is both an ancestor of A and a proper descendant of B has committed. If B is not an ancestor of A, then A is committed with respect to B if A is committed with respect to the least common ancestor of A and B in the transaction tree.

In Avalon programs, each data object performs its own synchronization and recovery. A transaction is guaranteed to be atomic if all the objects it manipulates are *atomic objects*. Avalon provides a set of built-in atomic data types that resemble typical built-in types (e.g., arrays and records), but these data types guarantee atomicity as well. As discussed below, Avalon also provides primitives to assist programmers in implementing their own atomic types. Serializability and recoverability are implemented for the built-in atomic types by Camelot facilities such as locking protocols, new value/old value logging, and commitment protocols.

A novel aspect of Avalon is that when a transaction commits, it is assigned a timestamp generated by a logical clock [Lamport 78]. Atomic objects are expected to ensure that all transactions are serializable in the order of their commit timestamps, a property called *hybrid atomicity*. This property is automatically ensured by two-phase locking protocols [Eswaran 76], but additional concurrency and availability can be achieved by taking the timestamp ordering explicitly into account [Herlihy 85]. Avalon provides a built-in transaction identifier type **tid** to assist programmers in implementing their own hybrid atomic data types. The tid type provides a restricted set of operations that facilitates run-time testing of serialization orders and the state of transaction commitment. Section 3.1 describes this type in more detail, and Section 4.2 gives an example of how it can be used to implement a highly concurrent atomic FIFO queue.

A second novel aspect of Avalon is that programmers may define type-specific *commit* and *abort* operations for user-defined atomic data types. The system automatically applies commit or abort when transactions terminate. Section 3.2 gives guidelines to users on what properties user-defined commit and abort operations should preserve and explains how the system invokes them, in particular, in the presence of nested transactions.

3. Avalon Built-in Types

For our examples we assume some familiarity with C++, in particular its subclassing mechanism. We begin by describing the *tid* type and the *atomic* type, both of which are used to implement user-defined atomic objects.

3.1. Transaction Identifiers

Avalon provides a *tid* data type to assist the programmer to reason about transaction serialization and commitment.

```
class tid {
    ...                                  // hidden representation
public:
    tid();                               // constructor
    int operator==(tid&);                // equality
    int operator<(tid&);                 // serialized before?
    int operator>(tid&);                 // serialized after?
    int done();                          // committed to top level?
    friend int both(tid&,tid&);          // both committed to lca?
    friend tid& root();                  // returns root tid
};
```

A new tid is created by declaring a variable or by a call to *new*:

```
tid& t = *(new tid);
```

Rather than simply returning the calling transaction's identifier, the tid constructor creates and commits a (dummy) subtransaction, returning the subtransaction's tid to the parent. This alternative semantics was chosen because it is often convenient for a transaction to generate multiple tids ordered in the serialization order of their creation events.

The system's current knowledge about the transaction serialization ordering can be tested by the overloaded operators "<" and ">." For example, if the expression:

```
t1 < t2
```

evaluates to *true*, then if both transactions commit, t1 will be serialized before t2. Note that < induces a *partial* order on tids; as long as t1 and t2 are concurrent, both t1 < t2 and t2 < t1 will evaluate to *false*. Eventually, as transactions commit, t1 and t2 will become comparable.

To process commits and aborts it is often necessary to determine whether certain transactions have committed. The *both* operation tests whether two transactions have both committed to their least common ancestor. If the following expression evaluates to true,

```
tid::both(t1,t2)
```

then t1 and t2 will either both commit to the top level, or they will both abort, but one cannot commit without the other.

The *done* operation tests whether a transaction has committed to the top level. This operation is primarily used to discard unneeded recovery information.

The *root* operation returns the tid for the unique transaction at the root of the transaction tree. All "top-level" transactions are children of the root. The expression `t.done()` is equivalent to `tid::both(t,tid::root())`.

3.2. Atomic Objects

Atomic objects are derived from the following built-in type:

```
class atomic {
  protected:
    virtual void seize();
    virtual void release();
    virtual void pause();
  public:
    virtual void commit(tid& t);
    virtual void abort(tid& t);
};
```

Each atomic object has a short-term lock, similar to a monitor lock, used to ensure that concurrent operations do not interfere. The short-term lock is acquired by the *seize* operation and released by the *release* operation. The *pause* operation temporarily releases the lock, suspends the caller, and reacquires the lock before returning. Any changes made to the object while the lock is held will not be backed up to stable storage until sometime after the lock is released. A transaction's changes are guaranteed to be backed up before it commits. These three operations are *protected*, meaning that they are accessible only within the implementations of derived types. These operations are implemented by the Avalon system, not the programmer.

The *commit* and *abort* operations are used to process commits and aborts. These operations are provided by the programmer but are called by the Avalon run-time system. After a transaction commits (aborts), the system will apply the commit (abort) operation to every object operated upon by a descendant of that transaction. Log records for commit and abort operations are spooled to the log, but need not be forced to stable storage. When a server recovers from a crash, it detects and reschedules missing commit and abort operations.

As illustrated in the examples in Section 4, commit typically releases locks and discards recovery information and abort typically releases locks and discards tentative changes. When implementing these operations, programmers should follow the following guidelines: (1) Most important, these operations should be viewed as "benevolent side-effects" in the sense that they should affect liveness properties, but not safety. For example, delaying a commit or abort operation may delay other transactions (e.g., by failing to release locks), but it should never cause a transaction to observe an erroneous state. (2) These operations should be idempotent, since a crash at an inopportune moment may cause a commit or abort operation to be applied to an object more than once for the same transaction. (3) These operations should not interact with the transaction system by creating, committing, or aborting transactions, thus they should not invoke operations of other atomic objects.

3.2.1. The Commit Operation

When the system calls an object's commit operation, it supplies the tid of the committing transaction as an argument. Commit operations are applied in leaf-to-root order. For example, if nested transaction A.B.C operates on x, then the system will call x.commit(A.B.C), x.commit(A.B), and x.commit(A), in that order. Commit operations for sibling transactions are applied in serialization order. If A and B operate on x and A is serialized before B, then x.commit(A) is applied before x.commit(B). The order in which commit operations for a given transaction are applied to multiple objects is left unspecified.

3.2.2. The Abort Operation

When the system calls an object's abort operation, it supplies the tid of the aborting transaction as an argument. Abort operations are also applied in leaf-to-root order, and the order in which abort operations for a given transaction are applied to multiple objects is left unspecified.

4. User-Defined Atomic Objects

In this section we give two examples, two-phase locks and FIFO queues, to illustrate how Avalon primitives can be used and what typical user-defined commit and abort operations do.

4.1. Two-Phase Locking

Here we give a simple implementation of Moss's nested two-phase locking protocol [Moss 81]. This example is for demonstration purposes only; Avalon provides more efficient low-level support for two-phase locking.

Moss's rules are:

1. A lock is granted to the caller if it is held by an ancestor.
2. If a transaction commits, the lock reverts to the parent transaction.
3. If a transaction aborts, the lock reverts to whomever held it before.

Here is the class definition for lock objects. Notice that *lock* is a derived class from the base *atomic* class, which was defined in the previous section.

```
class lock: public atomic {
    tid_stack s;
public:
    lock();                          // Constructor
    void request();                  // Acquire lock
    void commit(tid&);               // Release on commit
    void abort(tid&);                // Release on abort
};
```

The lock is represented by the *tid_stack* data type, which implements a stack of tid's.

The *request* operation returns after acquiring the lock. It waits until the transaction holding the lock is committed to an ancestor of the caller, and then pushes the caller onto the stack before returning.

```
void lock.request() {
    tid who = *(new tid);            // tid for this operation
    this->seize();                   // begin critical region
    for (;;) {                       // do forever...
        if (s.top() < who) {         // if serialized before ...
```

```
            s.push(who);          // push self onto stack ...
            this->release();      // leave critical region ...
            return;               // and return.
        };
        this->pause();            // otherwise wait and retry
    };
};
```

The *commit* operation reclaims storage by discarding redundant tid's.

```
void lock.commit(tid& who) {
    this->seize();                // enter critical region
    while (s.size() > 0 &&        // pop superfluous tid's
        tid::both(who,s.top()))
            s.pop();
    s.push(who);                  // push self back onto stack
    this->release();              // leave critical region
};
```

The *abort* operation also discards superfluous tid's, restoring the lock to whomever held it last.

```
void lock.abort(tid& who) {
    this->seize();                // enter critical region
    while (s.size() > 0 &&        // pop superfluous tid's
        tid::both(who,s.top()))
            s.pop();
    this->release();              // leave critical region
};
```

4.2. FIFO Queue

Here we give a more complex example— a highly concurrent FIFO queue. Our implementation is interesting for two reasons. First, it supports more concurrency than commutativity-based concurrency control schemes such as two-phase locking. For example, it permits concurrent Enq operations, even though Enq's do not commute. Second, it supports more concurrency than any locking-based protocol, because it takes advantage of state information. For example, it permits concurrent Enq and Deq operations while the queue is non-empty.

4.2.1. The Representation

Information about Enq invocations is recorded in the following struct:

```
struct enq_rec {
    tid enqr;
    item* what;
    enq_rec(tid& t, item* x) {enqr = t; what = x;};
};
```

The *enqr* component is a tid generated by the enqueuing transaction, the *what* component is a pointer to the enqueued item, and the last component defines a constructor operation for initializing the struct.

Information about Deq invocations is recorded similarly:

```
struct deq_rec {
    tid deqr;
    tid enqr;
    item* what;
```

```
        deq_rec(tid& d, tid& e, item* x)
            {deqr = d; enqr = e; what = x;};
};
```

The queue is represented as follows.

```
class queue: public atomic {
    stack deqd;
    pqueue enqd;
public:
    queue();                          // Create empty queue
    void enq(item*);                  // Enqueue an item
    item* deq();                      // Dequeue an item
    void commit(tid&);                // Called on commit
    void abort(tid&);                 // Called on abort
};
```

The *deqd* component is a stack of deq_rec's used to undo aborted Deq operations. The *enqd* component is a *partially ordered queue* of enq_rec's, ordered by their *enqr* fields. A partially ordered queue provides operations to enqueue an enq_rec, to test whether there exists a unique oldest enq_rec, to dequeue it if it exists, and to keep and discard all enq_rec's committed with respect to a particular tid.

This implementation satisfies the following representation invariant: First, an item is either "enqueued' or "dequeued," but not both: the enq_rec <tid, item> is in the *enqd* component if and only if for all x the deq_rec <x, tid, item> is not in the *deqd* component. Second, the stack order of two items mirrors both their enqueuing order and their dequeuing order: if deq_rec d1 is below deq_rec d2 in the *deqd* stack, then d1.*enqr* < d2.*enqr* and d1.*deqr* < d2.*deqr*. Finally, any dequeued item must previously have been enqueued: if d is a deq_rec, then d.*enqr* < d.*deqr*.

4.2.2. The Operations

The conditions under which Enq and Deq operations are allowed to occur are as follows. A transaction A may dequeue an item if (1) the most recent transaction to execute a Deq is committed with respect to A, and (2) there exists a unique oldest element in the queue whose enqueuing transaction is committed with respect to A. A may enqueue an item if the last item dequeued was enqueued by a transaction committed with respect to A.

Given these conditions, Enq is implemented as follows:

```
void queue.enq(item* x) {
    tid& who = *(new tid);
    this->seize();
    for (;;) {
        if (deqd.size()==0 || deqd.top().enqr < who) {
            enqd.enq(enq_rec(who,x));
            this->release();
            return;
            };
        this->pause();
    };
};
```

Enq checks whether the item most recently dequeued was enqueued by a transaction uncommitted with respect to A. If not, the current stamp and the new item are inserted in *enqd*. Otherwise, the transaction releases the short-term lock and tries again later.

Deq is implemented as follows:

```
item* queue.deq() {
    tid& who = *(new tid);
    this->seize();
    for (;;) {
        if (deqd.size()==0 || deqd.top().deqr < who) {
            if (enqd.can_deq() && enqd.oldest().when < who) {
                enq_rec e = enqd.deq();
                deqd.push(deq_rec(who,e.when,e.what));
                this->release();
                return e.what;
            };
        };
        this->pause();
    };
};
```

Deq tests whether the most recent dequeuing transaction has committed with respect to the caller, and whether *enqd* has a unique oldest item. If the enqueuing transaction has committed with respect to the caller, it removes the item from *enqd* and records it in *deqd*. Otherwise, the caller releases the short-term lock, suspends execution, and tries again later.

Commit is implemented as follows:

```
void queue.commit(tid& who) {
    if (! who.done()) return;
    this->seize();
    if (deqd.size() > 0 && deqd.top().deqr < who) deqd.reset();
    this->release();
};
```

When a top-level transaction commits, it discards deq records no longer needed for recovery. (The representation invariant ensures that all deq_rec's below the top are also superfluous, and can be discarded.)

Abort has more work to do:

```
void queue.abort(tid& who) {
    this->seize();
    while (deqd.size() > 0) {
        deq_rec d = deqd.top();
        if (d.deqr < who) {
            enqd.enq(enq_rec(d.enqr,d.what));
            d = deqd.pop();} else break;
    };
    enqd.discard(who);
    this->release();
};
```

Abort undoes every operation executed by a transaction committed with respect to the aborting transaction. It interprets *deqd* as an undo log, popping records for aborted operations, and inserting the items back in *enqd*. Abort then flushes all items enqueued by the aborted transaction and its descendants.

5. Related Work and Discussion

Transactions have been a primary focus in the context of both distributed and centralized data bases [Bernstein 81, Eswaran 76, Gray 78, Lindsay 79]. Reed [Reed 83] and Moss [Moss 81] have additionally proposed synchronization mechanisms for nested transactions. Several research projects have chosen transactions as the foundation for constructing reliable general-purpose distributed programs, including Argus [Liskov 83], Clouds [McKendry 84], and TABS [Spector 85]. Of these projects, however, only Argus has addressed the linguistic aspects of the problem.

On the other hand, other distributed programming languages such as CSP [Hoare 78], SR [Andrews 81], Linda [Gelernter 85], Nil [Strom 83], and Ada [Dod 83], have no direct support for transactions, and in particular for commit and abort processing of data objects. Many of these languages, e.g., CSP and Linda, also suffer from weaker type systems and have fewer abstraction mechanisms than what Avalon provides.

Avalon's model of computation resembles that of Argus and thus many Avalon features resemble those of Argus, e.g., servers are comparable to Argus guardians, nested transactions have the same semantics, and locking rules are similar. The principal way in which Avalon differs from Argus is in how it supports the implementation of user-defined atomic data objects. As described by Weihl and Liskov [Weihl 85], Argus does not provide explicit transaction identifiers, nor does it provide explicit commit and abort operations. In Avalon, explicit transaction identifiers are needed to support hybrid atomicity. Argus, by contrast, is based on a local atomicity property called *strong dynamic atomicity* [Weihl 84], which permits less concurrency, although it requires simpler run-time support. The desire and need for explicit control over commit and abort processing have been noted previously by Greif et al. [Greif et al. 86].

We chose to use C++, rather than C, to gain the advantages of strong type-checking, syntactic support for data abstraction, operator overloading, and subtyping, all of which we rely on. One problem we encountered early in our Avalon design for C++ was the lack of support for exception handling. We have since designed our own exception handling mechanism compatible with C++.

In summary, Avalon is a set of linguistic constructs that extend the capability of existing programming languages by directly supporting transactions and user-defined atomic objects. Users are relieved from the burden of doing low-level system activities such as locking and managing stable storage, and instead can concentrate of the logic required of their application. At the same time, however, they are given enough flexibility to exploit the semantics of their applications to increase their programs' efficiency, concurrency, and fault-tolerance.

Acknowledgments

The authors are grateful to Alfred Spector and Eric Cooper for their advice and assistance. We would also like to thank members of the Avalon group, Stewart Clamen, David Detlefs, David Waitzman, and Karen York, and members of the Camelot group, Dan Duchamp, Jeffrey Eppinger, and Dean Thompson.

References

[Andrews 81] G.R. Andrews.
Synchronizing Resources.
ACM Transactions on Programming Languages and Systems 3(4):405-430, October, 1981.

[Bernstein 81] P.A. Bernstein and N. Goodman.
A survey of techniques for synchronization and recovery in decentralized computer systems.
ACM Computing Surveys 13(2):185-222, June, 1981.

[Dod 83] Dept. of Defense.
Reference manual for the ADA programming language.
1983.
ANSI/MIL-STD-1815A-1983.

[Eswaran 76] K.P. Eswaran, J.N. Gray, R.A. Lorie, and I.L. Traiger.
The Notion of Consistency and Predicate Locks in a Database System.
Communications ACM 19(11):624-633, November, 1976.

[Gelernter 85] D. Gelernter.
Generative Communication in Linda.
ACM Transactions on Programming Languages and Systems 7(1):80-112, January, 1985.

[Gray 78] J.N. Gray.
Notes on Database Operating Systems.
Lecture Notes in Computer Science 60.
Springer-Verlag, Berlin, 1978, pages 393-481.

[Greif et al. 86] I. Greif, R. Seliger, and W.E. Weihl.
Atomic Data Abstractions in a Distributed Collaborative Editing System.
In *Proceedings of the 13th Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 160-172. January, 1986.

[Herlihy 85] M.P. Herlihy.
Comparing How Atomicity Mechanisms Support Replication.
In *Proceedings of the 4th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing.* April, 1985.

[Hoare 78] C.A.R. Hoare.
Communicating sequential processes.
Communications of the ACM 21(8):666-677, August, 1978.

[Lamport 78] L. Lamport.
Time, clocks, and the ordering of events in a distributed system.
Communications of the ACM 21(7):558-565, July, 1978.

[Lindsay 79] B.G. Lindsay et al.
Notes on Distributed Databases.
Technical Report RJ2571, IBM San Jose Research Laboratory, July, 1979.

[Liskov 83] B. Liskov, and R. Scheifler.
Guardians and actions: linguistic support for robust, distributed programs.
Transactions on Programming Languages and Systems 5(3):381-404, July, 1983.

[McKendry 84] M.S. McKendry.
Clouds: A Fault-Tolerant Distributed Operating System.
IEEE Tech. Com. Distributed Processing Newsletter 2(6), June, 1984.

[Moss 81] J.E.B. Moss.
Nested Transactions: An Approach to Reliable Distributed Computing.
Technical Report MIT/LCS/TR-260, Laboratory for Computer Science, April, 1981.

[Reed 83] D.P. Reed.
Implementing atomic actions on decentralized data.
ACM Transactions on Computer Systems 1(1):3-23, February, 1983.

[Spector 85] A.Z. Spector, J. Butcher, D.S. Daniels, D.J. Duchamp, J.L. Eppinger, C.E. Fineman, A. Heddaya, and P.M. Schwarz.
Support for Distributed Transactions in the TABS prototype.
IEEE Transactions on Software Engineering 11(6):520-530, June, 1985.

[Spector et al. 86] A.Z. Spector, J.J. Bloch, D.S. Daniels, R.P. Draves, D. Duchamp, J.L. Eppinger, S.G. Menees, D.S. Thompson.
The Camelot Project.
Database Engineering 9(4), December, 1986.
Also available as Technical Report CMU-CS-86-166, Carnegie Mellon University, November 1986.

[Steele 84] G. Steele Jr.
Common LISP.
Digital Press, 1984.

[Strom 83] R.E. Strom and S. Yemini.
NIL: An Integrated Language and System for Distributed Programming.
In *SIGPLAN '83 Symposium on Programming Language Issues in Software Systems.* June, 1983.
Also an IBM research report (RC 9499 (#44100)) from April 1983.

[Stroustrup 86] B. Stroustrup.
The C++ Programming Language.
Addison Wesley, 1986.

[Weihl 84] W.E. Weihl.
Specification and implementation of atomic data types.
Technical Report TR-314, Massachusetts Institute of Technology Laboratory for Computer Science, March, 1984.

[Weihl 85] W.E. Weihl, and B.H. Liskov.
Implementation of resilient, atomic data types.
ACM Transactions on Programming Languages and Systems 7(2):244-270, April, 1985.