Reliable Servers:
Design and Implementation in Avalon/C++

Richard Allen Lerner

September 1988
CMU-CS-88-177 3

Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Avalon/C++ is a programming language that supports the construction of reliable programs consisting of a set of servers communicating over a network. It provides high-level language support for user-defined data types with customized synchronization and fault-tolerance properties. These data types are encapsulated in servers, and accessed through exported server operations. Avalon/C++ greatly simplifies the programming of these servers by hiding the distributed nature of a server from both the implementor and callers of a server. Avalon/C++ exploits the similarity of servers and classes, making server definition and use look like that for C++ classes. A detailed description is given of a typical Avalon/C++ server, the catalog server, which is used by clients to locate servers.

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4976(Amendment 20), under contract F33615-87-C-1499 monitored by the Avionics Laboratory, Air Force Wright Aeronautical Laboratories, Wright-Patterson AFB.

This paper has also been published in the proceedings of the "1988 International Symposium on Databases in Parallel and Distributed Systems", Austin, TX, December 1988.

Introduction

A *distributed system* consists of multiple computers (called *nodes*) that communicate through a network. Distributed systems are typically subject to several kinds of failures: nodes may crash, perhaps destroying local disk storage, and communication may fail, via lost messages or network partitions. Writing reliable programs for distributed systems is difficult and has been the subject of many research projects. Avalon/C++ is the result of one such project. Avalon/C++ is a programming language that supports the construction of reliable programs consisting of a set of servers communicating over a network. It provides high-level language support for user-defined data types with customized synchronization and fault-tolerance properties. These data types are encapsulated in servers, and accessed through exported server operations. Avalon/C++ greatly simplifies the programming of these servers by hiding the distributed nature of a server from both the implementor and callers of a server. This paper explores the representation of servers in Avalon/C++.

In the first section, we give a brief description of Avalon/C++ and its run-time environment. A full description of Avalon's **server** type and its use is the topic of Section 2. The following sections discuss the various mechanisms used by the server representation. Section 3 describes how clients locate running servers. Section 4 describes a typical Avalon/C++ server, the catalog server, which is used in locating servers. A complete description of a user-defined concurrent data type used by the catalog server is given in Section 5. The last section details the current status and future plans for Avalon/C++, and discusses related work.

1. Overview of Avalon/C++

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, transaction-consistent, and permanent. *Serializability* means that transactions appear to execute in a serial order. *Transaction-consistency* ("all-or-nothing") means that a transaction either succeeds completely and *commits*, or *aborts* and has no effect. *Permanence* means that the effects of a committed transaction survive failures.

Avalon/C++ provides transaction semantics via *atomic objects*. Atomic objects ensure the serializability, transaction-consistency, and permanence of the transactions that use their operations. All objects used by transactions must be atomic. Avalon/C++ provides a collection of built-in atomic types; users may define their own atomic types by subtyping the built-in types. Avalon/C++ includes a variety of primitives (not discussed here) for creating transactions in sequence or in parallel, and for aborting and committing transactions. Included with these primitives is a mechanism to handle transaction aborts as exceptions.

The Avalon/C++ class hierarchy is made up of three classes, **recoverable**, **atomic**, and **subatomic**, arranged in an inheritance tree, where **atomic** and **subatomic** both inherit from **recoverable**. The most basic class in our hierarchy is **recoverable**. It ensures *permanence:* after a crash, a recoverable object will be restored to a state that reflects all operations performed by transactions that committed before the crash. **Atomic** is a subclass of **recoverable**, specialized to provide two-phase read/write locking and automatic recovery.

Locking is used to ensure serializability, and the automatic recovery mechanism ensures transaction-consistency. The third, and perhaps most interesting, base class in the hierarchy is `subatomic`. Like `atomic`, `subatomic` ensures atomicity. While `atomic` provides a quick and convenient way to define new atomic objects, `subatomic` provides primitives to give programmers more detailed control over their objects' synchronization and recovery mechanisms. This control can be used to exploit type-specific properties of objects to permit higher levels of concurrency and more efficient recovery. [Weihl&Liskov 85, Herlihy&Wing 87].

An Avalon/C++ program consists of a set of servers, each of which encapsulates some data objects. Each server provides concurrent access to a set of objects through exported operations. A server resides at a single physical node, but each node may be home to multiple servers. Rather than sharing data directly, servers communicate by calling one another's operations. An operation call is a remote procedure call (RPC) with call-by-value transmission of arguments and results. A server's objects are *stable*, that is, they survive crashes.

Avalon/C++ uses the Camelot system [Spector et al. 86] to handle operating-system level details of transaction management, inter-node communication, commit protocols, and automatic crash recovery. It also uses light-weight processes (threads), provided by the Mach/Unix [Accetta et al. 86] operating system, to support concurrency within a server.

2. Avalon's Server Type

The goal of Avalon's representation of servers is to hide the complexity associated with invoking and processing operation requests. A server is an encapsulation of some data objects with operations to access the data much like an ordinary C++ class. Avalon exploits this similarity, making servers look like classes, both from the point of view of the server implementor and the server client. Before describing how servers are defined and used in Avalon, it is useful to explain how a server and client are implemented in the underlying machine.

A server in the underlying machine is a program, running on a single node, that accepts RPC messages containing operation invocation requests. The server's receiver, running as a light-weight process in the server (a thread), unpacks the operation's parameters from the message and calls the server function that implements the operation. When a server function returns, the results are packaged into the return message sent to the client. To allow concurrency within the server, multiple threads are used to service the requests. Whenever a request arrives, a new thread is started to handle it. A server may contain additional threads to perform background processing independent of operation requests.

A client invokes a server's operations by sending a request to the server and waiting for a reply. If it cannot contact the server or times out waiting for the reply, the innermost transaction will abort, allowing Avalon's transaction abort handler mechanism to be invoked.

The `server` construct in Avalon hides all of the communication processing inherent in servers. A server is defined as a special C++ class. Like any other class, it contains some data declarations and operations which manipulate the data. The former are the data objects encapsulated by the server and the latter are the operations that the server exports.

Implementations of the operations are provided in the usual manner, as if the server were simply a class rather than a separate process. In particular, Avalon takes care of generating the code that initializes the process as a server, receives and unpacks RPC messages, calls the proper operation (in a new thread), and packages the results into a reply sent back to the client. To use a server, a client program simply includes the server definition and invokes operations on instances of the server, as with any other C++ class. From the client's perspective, the only difference between a sever and a normal class concerns how instances of the server are obtained. A simple server is shown in Figure 2-1.

```
server testS
{
    stable atomic_int val; // Protected Data
  public:                  // Constructor
    int(node n, path p, size s, int i):(n,p,s);
    int  get();            // Exported Op.
    void set(int i);       // Exported Op.
    void recover();        // (recovery)
    main();                // (background)
};
```

Figure 2-1: A simple server definition

For the remainder of the paper, the following definitions are used:

server, server definition:	the definition of the server. This looks like a C++ class definition with the keyword **server** replacing **class**[1]. Both the client and server programs include the same server definition.
server process:	the process running on a single node that services requests from clients for server operation invocations. A server process is described by a server definition and the accompanying operation implementations.
server object:	an instance of a server definition. A server object is the data structure in a client that represents a particular running server process of the appropriate type. When a new server object is created, a new server process is also created. Likewise, when a server object is destroyed, the server process is killed.
server reference:	a C++ reference to a server object. Clients generally use references to server objects, rather than the objects themselves. This allows multiple references to the same server object.
server type name:	the name used in the server definition. The server type name is used to declare server object and server reference variables. It is also used when locating a server to restrict the search to servers of the appropriate type.
server unique name:	the name that a server process uses to identify itself for the purposes of RPC communication. The underlying machine uses a global namespace for these names.

[1]A server may not be derived from another class since all servers are, in effect, derived from the class **server_root**. Also, no class or server may be derived from a server. Avalon does not define semantics for derived servers.

2.1. Servers from the client's perspective

From the client's perspective, an Avalon server is an instance of a server definition, a server object. A client invokes an operation on a server by calling a member function of a server object. Creating a new server object causes a new server process to be started. When a server object is deleted, the server is killed.

```
    // Start a new printer server
printS& p = *(new printS(...));

    // Locate an existing printer server
printS& p = (printS&) locate_server(...);

    // Invoke an operation
p.spool ("myfile.txt");
```

Figure 2-2: Example of Using Servers

For an Avalon program to make use of a server it must first obtain an instance of the appropriate server. As shown in the Figure 2-2, the client may either create a new server object, starting a new server process, or it may, with the `locate_server` function, obtain a reference to an *existing* server object representing a running server process.

The underlying machine associates an RPC communication address with a server's unique name. A name, however, is not necessarily the most convenient means of identifying a server. When there are multiple instances of a server, a client often does not know the unique name of the specific server to use. It may not care which instance to use, or it may wish to select the server based on some attributes particular to one instance of the server (e.g., the machine on which it is running, or the printer it is controlling). The Avalon library provides the `locate_server` function to allow selections of this sort. This function allows clients to locate a server object based on a list of attributes describing the server. The operation of this function is fully described in Section 3.

All servers inherit from the class `server_root`. This class provides the data and operations required by all servers. A constructor is provided to start a new server process when a client creates a new server object. When a server is created, the server's constructor calls this constructor with appropriate parameters. If a server does not provide its own destructor to gracefully kill a server process, `server_root` provides a destructor that sends a kill signal to the underlying machine to kill the server process whenever a client destroys a server object. The `server_root` also provides a place to store the information necessary for the client to communicate with the server process, including the unique name of the server process and its RPC address. `Server_root` actually provides a number of constructors, reflecting the varying amount of information which may be supplied in order to start any server. Common parameters to the constructors are: the node on which to start the server, an executable to use, and the amount of recoverable memory to allocate for the server. Except for these parameters, the programmer need not know about the `server_root` class.

When looking at a server definition, a programmer sees what looks like a class definition. Invoking an operation on the server is identical to invoking an operation on a C++ class (as shown in the third line of Figure 2-2). When a client invokes a server operation, however, Avalon takes over and a lot of behind-the-scenes action takes place.

Within a client, Avalon replaces the implementation of each server operation with code that packs the arguments into a message and initiates an RPC call to the server named by the unique name stored in the **server_root** component. When the RPC call returns, the results are unpacked and form the result of the client's call. Avalon makes similar replacements for the server's constructor and destructor to provide for starting and killing servers.

Since server objects are really just C++ objects with special operations, they can be manipulated in the same manner as other C++ objects. In particular, server objects and references to servers can be passed as parameters to and returned as values from functions.

2.2. Servers from the implementor's perspective

From a server implementor's perspective, a server is completely specified by its definition and member function implementations. A simple server definition is shown in Figure 2-1. The definition contains the following parts:

- data object declarations
- constructors
- exported operations
- private operations
- **main**
- **recover**

All of the data objects used by the server are declared in the server definition. These data objects are restored following a failure. It is important to note that, to be properly restored, the data objects must be derived from one of Avalon's three built-in classes **recoverable**, **atomic** or **subatomic**. Furthermore, they must be implemented to control concurrent manipulation. For example, the server in Figure 2-1 declares an **atomic** integer. The operations on this type[2], assignment and coercion to integer, are implemented to allow multiple readers or a single writer.

The exported operations list the parameters that Avalon must package up for the client and unpack for the server. The implementation of an operation provides the function that is called in the server process when a client invokes the operation. The communication code is generated by the Avalon compiler. Private operations are simply functions which can be called within the server from other member functions.

A server's constructor is very much like an operation. It defines the parameters that a client must use when creating a new server and provides code to execute when the server is started. The primary difference is that a constructor must also define the parameters that are sent to the underlying machine to start the server process. These parameters are specified as parameters to the parent's constructor (text to the right of the colon in Figure 2-1). When a client calls a server's constructor, the specified parameters are passed to the constructor for the

[2]**atomic_int** is one of a number of basic atomic types provided by the Avalon Library.

server_root class (described above) where they are given to the underlying machine. Once the server process is started, the client constructor code invokes the server's constructor operation in the same manner as any other operation.

The **main** member function provides a place for the implementor to put code which is executed as a background process when the server is started. This function can be used to provide code which needs to be run independently of operation invocations. A printer server, for example, could use **main** for the code to run the printer. **Main** must exist, even if empty, because Avalon uses the existence of a **main** implementation to determine that the current compilation is for a server, rather than just for a client.

The **recover** member function is optional, but provides a place for code that will be executed whenever the server is re-started after a failure.

3. The locate_server Function

In order for a client to invoke operations on a server, it needs to obtain a reference to its server object. Servers are identified in the underlying machine by a unique name. This name, in general, is not known by a client. Furthermore, if there are multiple instances of the server (e.g., printer servers for multiple printers), the client may wish to select a particular server based on some set of attributes. To provide this service, the Avalon library contains the function **locate_server**. This function takes the server's type name and an optional attribute list and returns a reference to a server object matching those attributes. Since **locate_server** is not specific to a particular server, it returns a reference to a **server_root** object. This should then be coerced to the appropriate server[3]. Figure 3-1 shows an example of the use of **locate_server**.

```
        // CMU printers are named after gems
attr_list attr;          // an attribute list
attr.push("printer", "pearl");

        // Find the desired server
printS& p = (printS&)locate_server("printS",attr);

        // Spool a file onto it
if (&p != NULL)
    p.spool (filename);
```

Figure 3-1: Using **locate_server**

Since it is expected that the result will be coerced to the desired server type, it is crucial that **locate_server** looks only for servers of the appropriate type. Thus, **locate_server** explicitly requires the server's type name as its first parameter. If any instance of the named server will do, the attribute list may be empty.

Attribute lists are currently very simple structures. They consist of a list of name-value pairs. A future enhancement may allow attribute expressions rather than simply attribute lists. These

[3]This works since the server class definition generated by Avalon for a client does not add any data objects and does not use virtual functions.

would allow boolean expressions on attribute values.

Avalon presents the model that creating a server object starts a new server. When locating a server, however, it is necessary to physically create a server object in the client. Hiding this creation is a secondary purpose for the **locate_server** function. When **locate_server** is called, it uses an Avalon server, the catalog server, to find the unique name of the desired server and uses a special constructor (provided by **server_root**) to create and initialize a server object representing the server. This special constructor does not start a new server. It simply allocates the structure. **Locate_server** then uses a reference to this object as its return value.

Although clients, in general, should not delete server objects obtained from **locate_server** (killing the server), the server objects created by **locate_server** have an internal flag set that inhibits the actual de-allocation of the object if deleted. Thus, a client is allowed to lookup a server, delete it, and look it up again. The final result would be a valid reference to a server object. The object, however, may represent a server that is not running. This would become apparent if a server operation were called (aborting the innermost transaction).

4. The Catalog Server

The catalog server is part of the Avalon runtime environment. It provides a repository of information about running servers. It is the job of the catalog server to maintain a mapping of server attributes to unique names, and to service lookup requests.

The catalog server is a good example of a typical Avalon server. It must reliably maintain the server-attribute mappings and provide concurrent access to this database. The server, as shown in Figure 4-1, provides operations to check in attributes for a new server, modify attributes, and to locate a server that matches a given attribute list.

There is exactly one catalog server. Since it is expected to be used relatively infrequently, we do not expect it to be a bottleneck. However, if experience shows otherwise, we may decide to run one per node in future versions of Avalon.

When a server starts, it must check in its attributes. The required attributes (i.e., type name, unique name and, node) are checked in by the initialization code for starting a server. The implementor of a server may provide additional information in the constructor code for the server. For example, the printer server should add the name of the printer it is servicing. When a client wants to locate a server, the **locate_server** function adds the server's type name (the first parameter) to the given attribute list, and calls the catalog operation **name**. To avoid boot-strapping problems, Avalon ensures that all clients have a reference to the catalog server, which has a fixed unique name.

Figure 4-1 shows the definition of the catalog server. When a new entry is created (with **check_in**) it is given a unique id. This id can later be used to look at and modify the attributes for the entry. In addition to **check_in**, the server provides a number of operations to set and query the attributes of a server as well as two operations to find a server based on an attribute list. The first form, **find**, returns the unique id of the described entry. The second form, **name**,

```
server catalog {
    stable    atomic_cat_hashtable_ptr servers;
    stable    atomic_int next_id;
  public:
    catalog(node n, path p, size s) : (n,p,s);
    int       check_in (attr_list alist);        // returns an id
    void      remove   (int id);
    void      set_attributes   (int id, attr_list new_alist);
    void      set_attribute    (int id, xString attribute, xString new_value);
    void      remove_attribute (int id, xString attribute);
    attr_list get_attributes (int id);
    xString   get_attribute   (int id, xString attribute); // returns value
    int       find (attr_list alist);            // returns an id
    xString   name (attr_list alist);            // returns unique name
    void      main ();
};
```

Figure 4-1: The Catalog server definition

returns the value of the selected entry's `unique_name` attribute. This form is equivalent to a `find` followed by a `get_attribute`, and is provided since this is a common operation.

Currently the catalog server is implemented using atomic hash tables. An entry in the catalog is created for each server. An entry is represented by a small hash table mapping the server's attribute names to values. The entries are kept in the hash table `servers`, keyed by id. The next section describes the atomic hash table implementation.

5. Atomic Hash Table

The main data type used by the catalog server is a hash table. Since the server's interface operations are expected to run concurrently, concurrent access to the hash tables must be allowed. This section describes an implementation derived primarily from the class `atomic`.

The hash table is implemented as a fixed length array containing pointers to a linked list (the buckets) of entries which hash to the bucket. Each bucket in the array is an `atomic` object consisting of three fields: `key`, `value`, and `next`. The object representing the hash table itself is derived from `recoverable`.

The hash table provides the following operations:

`Lookup`:	Given a `key` return the `value`.
`Insert`:	Given a `key` and a `value`, add a new entry, or return FALSE if the entry already exists.
`Alter`:	Given a `key` and a `value`, modify an existing entry, or return FALSE if the entry is not found.
`Remove`:	Given a `key`, remove the entry from the hash table, or return FALSE if the entry is not found.

5.1. Synchronization Conditions

Concurrency is controlled at the bucket level with the read/write locking provided by Avalon's **atomic** class. This is the standard multiple readers/single writer concurrency control. At the hash table level, there may be multiple writers as long as each writer is dealing with a unique bucket. The hash table proper does not need locking since, with this implementation, there are never any changes to the hash table structure. All changes occur within the buckets. The operations work as follows:

Lookup: acquires a read lock on the appropriate bucket and searches the list of entries in the bucket for the desired key.

Insert, **Alter**, **Remove**:
acquires a read lock on the appropriate bucket and searches for the indicated entry. If the entry exists (does not exist), it returns FALSE. Otherwise, it acquires a write lock on the bucket and adds (modifies or removes) the entry.

A successful **insert**, **alter**, or **remove** blocks all readers and writers of that bucket. If unsuccessful, they block only other writers. A **lookup** blocks only writers. In all cases, once an operation has returned a result, no other transaction will be able to make a change that would contradict the result, until the first transaction commits or aborts. The bucket remains locked until the end of the transaction.

5.2. Implementation

The hash table is built from three types of structures. The top-level structure, shown in Figure 5-1, is the **atomic_hashtable** class. This structure contains an array of buckets and a size. The size of the array is set when the hash table is constructed. This class is only **recoverable** since none of the operations change the contents except through operations on the buckets. Since there are no concurrency problems, there is no need for concurrency control. The basic classes do not specify the types for the hash table keys and values. It is expected that a programmer will specialize these classes with the appropriate types. The functions **hash_tag** and **newentry** are virtual and are expected to be provided by the specialized hash table class. **Hash_tag** provides a key type specific routine to generate an integer from a key. This function is called by **atomic_hashtable**'s private **hash** function. The **newentry** function is called whenever a new **hashentry** is needed. The specialized version of this function allocates an instance of the appropriate specialization of **hashentry** rather than the generic one.

Each bucket in the hash table is of class **atomic_bucket**. This class represents an atomic set implemented by a linked list of **hashentrys**. The class inherits from **atomic** and each operation acquires either a read lock or a write lock on the bucket, protecting modifications to the **head** pointer and the linked list. The linked list elements are protected since all modifications can only occur as a result of one of the operations on **atomic_bucket**.

```
class atomic_hashtable : public recoverable {
    atomic_bucket* buckets;                       // buckets array
    int size;

    int hash (const g_ptr tag);                   // returns a bucket index
    virtual int hash_tag (const g_ptr tag);       // returns an int given a key

  public:
                                                  // creates a new entry struct
    virtual hashentry* newentry (g_ptr tag, g_ptr data, hashentry* nxt);
    atomic_hashtable (int);                       // constructor
    ~atomic_hashtable ();                         // destructor

    const g_ptr lookup (const g_ptr tag)
        { return buckets[hash(tag)].lookup (tag); }
    bool insert (g_ptr tag, g_ptr data)
        { return buckets[hash(tag)].insert (tag, data, this); }
    bool alter  (g_ptr tag, g_ptr data)
        { return buckets[hash(tag)].alter (tag, data); }
    bool remove (const g_ptr tag)
        { return buckets[hash(tag)].remove (tag); }
};
```

Figure 5-1: The top-level atomic hashtable class

```
class atomic_bucket : public atomic {
    hashentry* head;       // linked list of entries
  public:
    atomic_bucket ();
    ~atomic_bucket ();
    const g_ptr lookup (const g_ptr tag);
    bool insert (g_ptr tag, g_ptr data,
                 atomic_hashtable* ht);
    bool alter  (const g_ptr tag, g_ptr data);
    bool remove (const g_ptr tag);
    bool is_empty  ()
        { return (head == NULL); }
};
```

As long as a type specific derivation of **atomic_hashtable** provides a **newentry** function which creates entries of an appropriate subtype, the **atomic_bucket** class is type-independent. No specializations of this class are needed.

The individual entries are kept in a linked list of objects of class **hashentry**. This class inherits from **recoverable** rather than **atomic** since it is protected by the **atomic** list head. A **hashentry** contains three fields: the recoverable key and value and a next pointer. The fields **key** and **value** are **recoverable*** rather than **void*** since all objects pointed to from within a **recoverable** structure must themselves be **recoverable** to allow recovery and undo.

```
class hashentry : public recoverable {
  protected:
    recoverable*    key;
    recoverable*    value;
    hashentry*      next;
  public:
    hashentry ();
    hashentry* get_next ();
    void set_next (hashentry* nxt);

    virtual ~hashentry ();
    virtual bool test_key (const g_ptr tag);
    virtual const g_ptr get_key ();
    virtual const g_ptr get_value ();
    virtual void set_value (g_ptr data);
};
```

To implement a hash table with specific types for the key and value, a new class is derived from this class, overriding the constructor and access functions to allocate and access the values appropriately.

The following code implements the **atomic_bucket lookup** operation.

```
const g_ptr atomic_bucket::lookup(const g_ptr tag)
{
    (*this).read_lock();
    hashentry* cur_ent = head;

    while (cur_ent != NULL)
    {
        if ((*cur_ent).test_key (tag))
            return (*cur_ent).get_value();
        cur_ent = (*cur_ent).get_next();
    }

    // NOT IN LIST

    return NULL;
}
```

It first acquires a read lock on the bucket. If another transaction already has a write lock on this bucket, the operation will suspend until it can acquire the read lock. Once locked, it scans the list. For each entry in the list, the entry's **test_key** function is called to determine if it is the desired entry. If found, the value of the entry is returned.

The implementation of the **atomic_bucket insert** operation, shown below, first checks that the entry does not exist, and if not, adds it.

```
bool atomic_bucket::insert (g_ptr tag, g_ptr data,
                            atomic_hashtable* ht)
{
    (*this).read_lock();
    hashentry* cur_ent = head;

    while (cur_ent != NULL)
    {
        if ((*cur_ent).test_key(tag))
            return FALSE;          // Already in list
        cur_ent = (*cur_ent).get_next();
    }

    // Not in list, Insert it

    (*this).write_lock();
    pinning()
        head = (*ht).newentry (tag, data, head);
    return TRUE;
}
```

`Insert` takes a pointer to the hash table so that it can use the correct user-defined `newentry` function provided by the type specific class derived from `atomic_hashtable`. The `pinning` statement is part of the Avalon extensions to C++ for crash recovery and abort handling. The `pinning` statement tells the underlying machine that a `recoverable` object is being modified, in this case, the bucket structure. When a pinning block is entered, the contents of the object are saved. When the block exits (for any reason) the changes to the contents are logged. If a transaction aborts, the underlying machine finds all of the `recoverable` objects which were changed and reverts them back to their original value. If a fault occurs, all of the `recoverable` objects will be restored to their value as of the last committed transaction.

If two transactions attempt to simultaneously insert an entry in the same bucket, it is possible for deadlock to occur. Both may have read locks on the bucket inhibiting the other from getting a write lock. This is not a problem since the underlying machine detects deadlock and aborts one of the transactions, allowing the other to acquire the write lock.

5.3. More Concurrency - An Alternate Implementation

The `atomic` hash table implementation, as one might guess, does not provide as much concurrency as one might want. An obvious enhancement would allow multiple writers at the bucket level. Instead of locking the entire bucket, the `value` and `next` fields in the `hashentry` class could be `atomic`. The `insert` operation, for example, could scan the linked list, acquiring read locks on each `next` field until the end of the list or an entry with the same key was found. If the end of the list was found, it could acquire a write lock on the last `next` field and insert the new entry. The write lock would prevent other changes or lookup of the new entry, but would not affect lookup of other existing entries. Unfortunately, doing so makes the implementation much more complicated because the concurrency control mechanism is tied in with the data storage implementation. The trouble occurs in the following situation. One transaction has a read lock on the last `next` field from a failed `lookup`. A second transaction tries to insert a new entry. It is able to search the list to see if the entry already exists, but if it does not, when it tries to add the new entry onto the end of the list, it correctly waits for the first transaction's read lock to be released. Finally, a third transaction also tries to concurrently insert a new entry and waits for the read lock to be released. When the first transaction releases its read lock, the other two transactions, one after the other, gain the write lock and set the **next** field to point to their new entry and commit. The result is that only the new entry added by the last transaction to commit ends up in the list. To solve this problem, the inserting transactions must check, after they receive the write lock, whether they are still looking at the last entry or if a new entry (possibly with the same key) has been added. The problem is that normal read/write locking is fairly rigid. Furthermore, even if one were to solve these problems, full concurrency (blocking only if the keys are the same) would not be achieved. Instead, it would have properties such as for **remove**, "it is ok to lookup entries prior to the one removed, but not those after it." To really solve these problems, Avalon provides another class, **subatomic**, which allows more flexible concurrency control schemes. This is fully explained in [Detlefs et. al. 88]. A brief description follows.

Avalon's **subatomic** class provides short term locks for consistency control instead of read/write locks, comparison of transaction identifiers for concurrency control, and user-defined commit and abort functions. The basic approach is:

1. Obtain a short-term lock on your entire data-structure.
2. Find the object to modify.
3. Use transaction identifiers to determine if you can make the change.
4. Make the change, marking it with your transaction identifier and saving the previous version.
5. Release the short term lock.
6. Clean up at the end of the transaction.
    a. If you commit, throw out the undo information
    b. If you abort, undo your changes.

The **subatomic** implementation of the hash table provides the same functionality as the **atomic** version, but with increased concurrency. The primary advantage of the **subatomic** implementation is that concurrency control, consistency protection, and recovery are separated from the data storage. Whereas the **atomic** implementation uses read/write locking to provide all of these properties, the **subatomic** implementation uses a lock table for determining concurrency conflicts, short-term locks for consistency protection, and **atomic** objects for data recovery. Since concurrency and consistency control are separated from data storage, the code manipulating the data structures is straightforward. While an **atomic** implementation would need to get the appropriate locks on each element of the linked list during a search, the **subatomic** implementation first checks for concurrency conflicts, gets a short-term lock on the data structure to ensure that no one else can access it, and finally scans through the list as in a non-concurrent implementation.

Another advantage of the **subatomic** approach is that it is more independent of the implementation used for the data storage and operations. The concurrency control is entirely independent of how the data is stored. One disadvantage to **subatomic** implementations is the added work for the implementor.

6. Status

We are currently implementing Avalon/C++ on IBM RT's and DEC MicroVaxen using Version 1.1 of C++ [Stroustrup 86]. The implementation takes the form of a preprocessor that transforms Avalon code to C++ code. We make extensive use of the Camelot system for low-level transaction support. Camelot, in turn, relies on the Mach operating system [Accetta et al. 86] for memory management, inter-node communication, and lightweight processes. At this time, all of the mechanisms described in this paper are in use. In the future, we hope to implement the `subatomic` version of the hash table and to do performance evaluation.

7. Acknowledgements

The initial idea of a catalog server was inspired by Argus's [Liskov&Scheifler 83] catalog subsystem. Jeannette Wing and Maurice Herlihy are responsible for much of the general design of Avalon/C++, as well as some of the introductory text. Also involved in the design were Dave Detlefs, who did much of the implementation, Stewart Clamen, Karen Kietzke, and Su-Yuen Ling. I also wish to thank Jeannette Wing and Barbara Staudt for their comments on the paper.

References

[Accetta et al. 86] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young.
Mach: A New Kernel Foundation for UNIX Development.
In *Proceedings of Summer Usenix*, July, 1986.

[Detlefs et. al. 88] D. Detlefs, M. P. Herlihy, and J. M. Wing.
Inheritance of Synchronization and Recovery Properties in Avalon/C++.
In *The Proceedings of the 21st Hawaii International Conference on System Sciences*, Kailua-Kona, Hawaii, Jan, 1988.
Also available as CMU-CS-TR-87-133, June 1987.

[Herlihy&Wing 87] M. P. Herlihy and J. M. Wing.
Avalon: Language Support for Reliable Distributed Systems.
In *The Proceedings of the 17th International Symposium on Fault-Tolerant Computing*, Pittsburgh, PA, July, 1987.

[Liskov&Scheifler 83]
B. Liskov and R. Scheifler.
Guardians and Actions: Linguistic Support for Robust, Distributed Programs.
ACM Transactions on Programming Language and Systems 5(3):382-404, July, 1983.

[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.

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

[Weihl&Liskov 85] W. E. Weihl and B. Liskov.
Implementation of Resilient, Atomic Data Types.
ACM Transactions on Programming Language and Systems 7(2):244-269, April, 1985.