Data Consistency in a Distributed Persistent Object System

Z. Wu — Department of Computing Science, University of Newcastle, UK

K. Moody and J. Bacon — Computer Laboratory, University of Cambridge, UK

R. J. Stroud — Department of Computing Science, University of Newcastle, UK

Abstract

A major issue in persistent systems is preserving data consistency in the presence of concurrency and failures. This paper presents a persistent system PC++ that takes an atomic data type approach to resolving this issue. Unlike existing systems, support for atomic data types in PC++ is implicit so that programmers are required to do very little extra work to make an object atomic. Programmers implement atomic data types as if for a sequential and reliable environment and specify the conflict relationship between object operations separately in a small, but expressive declarative language. The PC++ system will then automatically provide appropriate synchronisation and recovery code for atomic objects according to their conflict relation.

1 Introduction

The advent of high bandwidth local area ATM networks has transformed the potential of distributed computing systems. Current storage services are unable to meet the requirements of emerging application areas such as multimedia applications. As part of the OPERA project at Cambridge, a multi-service storage architecture (MSSA) for such applications has been designed and built [?, ?]. Performance tests are now being carried out.

In order to help programmers developing multimedia applications to make use of the MSSA, we have developed a persistent programming language PC++ [?]. PC++ extends C++ with persistent classes. A persistent class keeps all the features of a class in C++, but provides additional features that are important for multimedia applications. In particular, (1) PC++ provides a uniform method for applications to create and manipulate both temporary and persistent objects; (2) the application programmer may construct and manipulate object data structures whose components are stored across many special purpose servers within the MSSA; and (3) persistent objects can be located anywhere in a distributed system but are accessed as if they were local to the application.

Persistent data is usually constructed and shared by a community of users and may be distributed in different locations. A major issue for PC++, therefore, is preserving the consistency of data in the presence of concurrency and failures. PC++ uses atomic data types [?] to maintain data consistency. This enhances modularity by encapsulating synchronisation and recovery code in the implementation of shared objects and allows type-specific semantics to be used to allow greater concurrency between activities.

Atomic data objects that are shared by concurrent activities are responsible for ensuring their own consistency. PC++ takes an implicit approach to supporting atomic data types which requires programmers to do very little extra work to make an object atomic. Programmers implement atomic data types as if for a sequential and reliable environment and specify the conflict relationship between object operations separately in a small, but expressive declarative language. The PC++ system will then automatically provide appropriate synchronisation and recovery code for atomic objects according to their conflict relation.

In this paper we focus on the design and implementation of the mechanisms used by PC++ to preserve data consistency in the presence of concurrency and failures. In the next section we outline the support for data persistency in PC++. Section 3 introduces the notion of atomic data types and then shows how easily programmers can define both atomic data types and transactions in PC++. Section 4 and Section 5 focus on the implicit approach to implementing atomic data types. Section 4 describes in some detail the dual-level validation concurrency control method that is used by PC++ to provide local atomicity, whilst

Section 5 presents the method used to provide global atomicity in the presence of failures, in particular the distributed atomic commitment protocol. Comparisons with related work are given in Section 6, and Section 7 concludes the paper.

2 Data persistency

In this section, we introduce the data persistency model of PC++, and give an overview of its persistent object store.

2.1 Persistent objects

In PC++ an integrated persistence mechanism is provided: the persistent class. A persistent class keeps all the features of a class in C++, but its instances, called persistent objects, are allocated in persistent store and continue to exist after the program that created them has terminated.

In order to make a class persistent in PC++, the programmer need only place a preprocessor directive "persistent" in front of the class definition. For example, consider a bank account abstract data type that has an associated set of operations: *credit* money to an account, *debit* money from an account, and *check* the balance of an account. This data type can be defined in PC++ as shown in Figure ??.

```
persistent class Account
{
private:
        Money     balance;
public:
                  Account();
        Status    credit(Money);
        Status    debit(Money);
        Money     check();
};
```

Figure 1: The definition of Account

In PC++ a persistent object is named and protected at the system level by an *oid* (object identifier). An *oid* is a capability labeled with a principal and tagged with a type identifier. The *oid* can be used directly by programmers. PC++ also supports optional user-level names in the form of text strings.

For every persistent object, a *create* and an *invoke* operation are provided automatically by the system. A user program creates a persistent object by calling the *create* operation. The *create* operation can be called either with or without a string as the user-level name for the created object. In either case, the operation will return a system-level name, an *oid*, to the caller. To access an existing persistent object, a user program must pass either the *name* or the *oid* to the *invoke* operation to make the persistent object active. Then the program can manipulate it by calling any operations defined on it.

2.2 The multi-service storage architecture

PC++ stores both data and meta-data in the multi-service storage architecture (MSSA). The MSSA comprises an open, two-level hierarchy of servers (see Figure ??). The servers at the low level, the byte segment custodes (BSCs), manage storage media of any type, support a common byte segment abstraction and provide quality of service guarantees for acceptance and delivery of data. The architecture is such that the special purpose servers at the high level all employ the low level BSCs for storage.

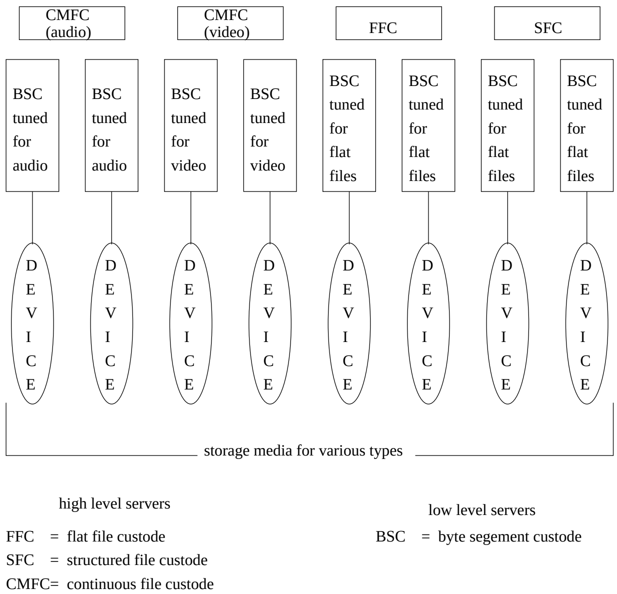

Figure 2: An example of an MSSA configuration

The servers at the high level provide storage abstractions for various types of stored object; for example, we have a flat file custode (FFC) for conventional files, a structured file custode (SFC) and a continuous media file custode (CMFC). There may be multiple instances of each type of custode.

Structured objects are represented on SFCs. An SFC supports only byte sequences and storage service identifiers (SSIDs) as primitive types but the constructors *sequence, record* and *union* may be used to create tree structures of arbitrary depth. A feature of SFC that is of importance to PC++ is that it allows its tree-structured objects to be accessed and manipulated atomically at sub-object level.

2.3 Object binding and data migration

A persistent object in PC++ is implemented as two parts: a *formal object* that is a C++ object and a *value object* that is an SFC object, see Figure ??. A value object is created when the *create* operation is called on a formal object. Value objects are stored on the SFC and can be shared by different users. Formal objects are transient; they are created by and are local to a user program, and they are destroyed automatically at the end of a program session.

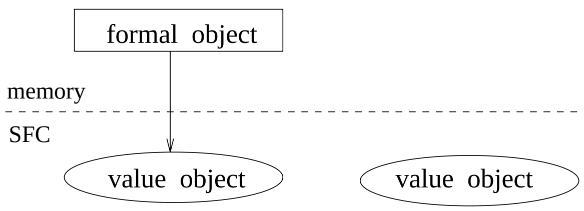

Figure 3: Persistent object composition

An essential property of a language with persistence is that objects in the object store can be manipulated using the same expression syntax as volatile objects. In order to execute such an expression there must first exist a binding between symbols in the program and objects in the object store.

In PC++ a value object stored in the SFC is bound to a formal object at run-time by specifying the *oid/name* of the value object as the argument to an *invoke* operation on the formal object. To ensure type consistency is not violated, it must be guaranteed that a value object can only be bound to a correct formal object. To achieve this, the identifier of the class of a persistent object is included in its *oid*. This can be checked against the class identifier of the formal object to which the value object is going to be bound.

Given that value objects reside on the SFC, a mechanism is needed to migrate the data in and out of user memory space during a program run. Data migration in PC++ does not happen with object binding, but with expression evaluation. Data can be migrated in and out of user memory space at any granularity. By moving in only the components that are needed to evaluate an expression, memory space and time can be saved. This allows programs to use very large persistent objects efficiently.

3 Data consistency

A major issue in persistent systems is preserving the consistency of data in the presence of concurrency and failures. PC++ uses distributed transactions to preserve data consistency.

3.1 The general approach

The implementation of transactions in PC++ is supported by the use of atomic data types. Instances of atomic data types, called atomic objects, are responsible for ensuring their own serialisability and recoverability. If all of the shared data objects accessed by concurrent transactions are atomic, then the transactions are guaranteed to be serialisable and recoverable. Serialisability and recoverability together are called atomicity. In the atomic data type approach, there are two kinds of atomicity. Atomicity of transactions is called *global atomicity* because it is a property of all of the transactions in the system. Atomicity for an object is called *local atomicity* because it deals only with the events involving the particular object [?].

Generally speaking, implementing atomic data types is a difficult task. This is because atomic data types define the behaviour of objects in a concurrent and unreliable environment. The implementation of an atomic data type needs: (1) to represent application information, synchronisation information and recovery information; (2) to implement synchronisation operations, recovery operations and object operations in terms of these representations; and (3) to specify the semantics of object operations. Application information is the data used to implement the functional requirements of an application; the other kinds of information are used for ensuring the consistency of application information in a concurrent and unreliable environment.

We divide approaches to implementing atomic data types into three classes: implicit, explicit and hybrid, according to whether the system or the programmer is responsible for implementing the synchronisation and recovery code. With an implicit approach, the programmer is only responsible for implementing the basic object operations as if for a sequential reliable environment with no concurrency or failure. The system is responsible for implementing the necessary synchronisation and recovery code using knowledge about the object's semantics provided by the programmer. With an explicit approach, the programmer is reponsible for implementing both the basic object operations and the synchronisation and recovery code. With a hybrid approach, the work of implementing the synchronisation and recovery code is shared by the system and the programmer. PC++ takes an implicit approach to implementing atomic data types, thereby ensuring a clean separation between the functional and non-functional aspects of atomic objects.

3.2 Defining atomic data types

Persistent objects in PC++ have the properties of atomic objects. However, programmers need to do very little extra work to make an object atomic due to the implicit approach. Programmers must first define the behaviour of objects for a sequential and reliable environment, and then need only specify the conflict relationship between operations in a simple but expressive declarative language.

When specifying the conflict relation in PC++, each operation is represented by its name and result. The result of an operation is simply characterised as either *failed* or *succeeded* (represented by OK), since usually only this distinction makes a significant difference to conflict relations. The first parameter of an operation can also be taken into account if appropriate. The relationship between two parameters can be classified as: $=$ or $\neq$.

For example, suppose there are two operations: *oper1* and *oper2* (they are not necessarily different). If *oper1* invalidates *oper2* in all cases, then this conflict can be represented as:

((*oper1, any*) (*oper2, any*)).

If *oper1* invalidates *oper2* only when their parameter is the same, then this conflict can be represented as:

((*oper1, any*) (*oper2, any*) =).

If *oper1* invalidates *oper2* only when their parameter is the same and both of them succeed, then this conflict can be represented as:

((*oper1, OK*) (*oper2, OK*) =).

We say that an operation p invalidates an operation q if the result of executing q on an object d might not be the same as the result of executing q after executing p on d.

For example, the *Account* abstract data type defined in Figure ?? could be made atomic by including a *conflict relation* part, as shown in Figure ??. The *conflict relation* of an object describes possible conflicts between object operations that would limit concurrency. In this example, the conflict relation describes four possible conflicts: a successful *credit* invalidates a successful *check*; a successful *debit* invalidates a successful *check*; a successful *debit* invalidates another successful *debit*; and a successful *credit* invalidates a failed *debit*. The interpretation of the conflict relation as a concurrency control policy is handled automatically by PC++. Thus, the application code that implements the *Account* class contains no explicit synchronisation code.

```
atomic class Account
{
private:
        Money     amount;
public:
                  Account();
        Status    credit(Money);
        Status    debit(Money);
        Money     check();
conflict relation:
        ((credit, OK) (check, OK))
        ((debit, OK) (check, OK))
        ((debit, OK) (debit, OK))
        ((credit, OK) (debit, failed))
};
```

Figure 4: The class Account

3.3 Transactions

Transactions are implemented in PC++ by a *Transaction* class with *begin_transaction*, *end_transaction* and *abort_transaction* operations. If an instance of an atomic data type is activated within a transaction, then PC++ will ensure that any operations performed on that object are atomic with respect to the transaction. The implementation of this is transparent to the application programmer who simply specifies where transactions begin and end.

Figure ?? shows how a programmer can construct a transaction that attempts to transfer some amount of money from one account to another. At first the transaction tries to debit £1000 from account *John*, and if the debit succeeds, it credits £1000 to account *Guang* and commits; otherwise, it aborts.

```
Account      John, Guang;
Transaction  T;

T.begin_transaction();
John.invoke("John");
Guang.invoke("Guang");
if (John.debit(1000) == Success)
  {
    Guang.credit(1000);
    T.end_transaction();
  }
else
    T.abort_transaction();
```

Figure 5: A transfer transaction

4 Providing local atomicity

In the next two sections we will describe the method used by PC++ to implement atomic data types.

4.1 Type-inheritance method

In PC++, the system rather than the programmer is responsible for providing the synchronisation and recovery operations for user-defined atomic objects. An essential issue therefore is how to integrate system-provided synchronisation and recovery with different kinds of object whilst still allowing type-specific concurrency control. PC++ resolves the issue by using the type inheritance technique of object-oriented programming. The method is quite straightforward. A special type, called *Scheduler*, is provided which implements a specific concurrency control protocol. Atomic data types are translated by the PC++ preprocessor into C++ classes that are derived from this special type and will thereby inherit the underlying concurrency control facility. Furthermore, if the definition of an atomic data type includes type-specific concurrency information about the semantics of its operations, this can be used by the inherited concurrency control mechanism to make synchronisation decisions based on these semantics.

A difficulty particular to an implicit approach is how the system can get the information necessary to implement synchronisation and recovery, because programmers should not be asked to provide it. PC++ overcomes the difficulty via the preprocessor. By preprocessing atomic type definitions, some code can be added to object operations so that the information necessary for synchronisation and recovery can be collected automatically when object operations are invoked by transactions.

4.2 The dual-level validation method

To provide implicit support for atomic data types, an appropriate concurrency control protocol must be used that can take advantage of operation semantics to increase concurrency and can be implemented independently from user-defined atomic data types. PC++ uses the *dual-level validation method* (DLV) [?] for providing concurrency control. The DLV is an optimistic method, *i.e.*, it allows transactions to execute without synchronisation, relying on commit-time validation to ensure serialisability.

The two levels of the DLV are concerned with two levels of the object architecture, logical and physical. The *logical level* is the set of abstract operations defined on an object. The *physical level* is the set of operations provided by the persistent storage system to manage primitive data objects: *create, delete, read* and *write*. Logical level validation ensures that a transaction that has used an object and is requesting a commit is serialisable with other transactions. Physical level validation ensures that the logical level object operations are elementary. Logical level validation can be done by making use of the semantics of the object so that greater concurrency can be achieved.

A transaction, in general, encloses operations on several objects. The sequence of operations of a transaction on a particular object forms the *component* of the transaction at that object. An execution of a transaction consists of two, three or four phases: a *read phase*, a *validation phase*, and possibly a *pending phase* and a *write phase*.

During the read phase, the transaction manager passes each operation enclosed in a transaction to the appropriate object. The object arranges immediate execution of the operation. If the invocation involves an update, this takes place on a local *shadow copy* of the physical sub-object. Each object maintains a record of which object operations have been performed by each transaction, and which physical (sub)objects have been read or written by each transaction.

The logical validation phase begins when the execution of a transaction reaches its end. During this phase, each object validates its component of the transaction and indicates *accepted* or *rejected*. The aim is to establish whether any of the invocations of the transaction have been invalidated by the invocations of concurrent transactions. Logical validation is done according to the semantics of the object.

Each accepted component of the transaction enters the *pending* phase. If every component of the transaction passes its logical validation, the transaction as a whole will be committed in the write phase, otherwise it will be aborted.

Transactions enter the write phase in the order defined by their timestamps which are assigned by the transaction manager at the end of the read phase. After entering the write phase, each transaction component is validated again by the object to check whether it can be accepted at the physical level. Physical validation is done by checking whether the version number of each physical object in the read set of a transaction component is still current. The purpose of this validation is to check whether the values read by the transaction are still up to date. If they are, the transaction component is committed by merging its shadow copies into the permanent state. Otherwise the shadow copies are discarded and the operations of the transaction component are re-executed.

It is worth pointing out that after a decision is made to commit a transaction, each component of the transaction enters the write phase independently. If one component fails to pass its physical validation in the write phase, only this component needs to be re-executed and no other components should be affected. Thus, PC++ limits the damage to the lowest level. This is a significant performance advantage that the DLV method has over many other methods.

4.3 Validation algorithms

The purpose of logical validation in the DLV method is to ensure that the concurrent execution of a set of transactions is equivalent to executing these transactions serially in some order. To do this, each transaction T_i is explicitly assigned a unique number t_i, called the *timestamp* of the transaction. The validation algorithm then ensures that there exists a serially equivalent schedule in which transaction T_i comes before transaction T_j whenever $t_i < t_j$. This can be guaranteed by the following validation condition [?]. For each transaction T_j with transaction number t_j, and for all T_i with $t_i < t_j$, one of the following three conditions must hold:

1. T_i completes its write phase before T_j starts its read phase.
2. The operation set of T_i does not invalidate the operation set of T_j, and T_i completes its write phase before T_j starts its write phase.
3. Neither the operation set of T_i invalidates the operation set of T_j nor the operation set of T_j invalidates the operation set of T_i, and T_i completes its read phase before T_j completes its read phase.

Although algorithms that implement all three conditions allow more concurrency, they require very complicated checking. Thus, the DLV method uses an algorithm that is an implementation of validation conditions 1 and 2 only. The first validation condition can be checked by recording the latest committed transaction's timestamp when a transaction starts. The second condition is validated by the following three checks (suppose transaction T_j is under validation):

- *Check 1*: For every transaction T_i that is *older* than T_j, and had not committed when T_j began, check whether the operation set of T_i invalidates the operation set of T_j; if it does, the validation fails.
- *Check 2*: For every transaction T_k that is in its pending phase and is *younger* than T_j, check whether the operation set of T_j invalidates the operation set of T_k; if it does, the validation fails.
- *Check 3*: Check whether any committed transaction T_k is *younger* than T_j; if any T_k is, the validation fails.

Check 1 and *Check 2* ensure that the first part of validation condition 2 holds. *Check 3* ensures that the second part of validation condition 2 holds.

By allowing users to specify whether an operation invalidates another operation according to type-specific semantics, and by using this specification to make validation decisions, the DLV method successfully integrates system-provided concurrency control with user-specified concurrency behaviour of objects.

4.4 Recording information

In order to validate and re-execute a transaction, the DLV method needs to record the events of each transaction in an *event table*. The preprocessor inserts some extra code in each object operation to record the necessary information. Thus, whenever an operation is invoked on an atomic object the operation's name, parameters and result are automatically recorded in the *event table* associated with the transaction that invoked the operation.

The state of an object in PC++ is represented by a physical object and is stored in the MSSA. Because the DLV method requires that an object can be accessed and locked at sub-object level, a special object manager in the MSSA, called the structured file custode (SFC), is used. The SFC allows a tree-structured object to be accessed, manipulated and locked at any level. Therefore, when making changes to a component of an object, only that component needs to be rewritten; no other component of the object is affected at all.

5 Global atomicity

In this section, we describe how to ensure the global atomicity of a system when objects use the DLV method for providing local atomicity, in both the absence and the presence of failures.

5.1 System model

The architecture of the distributed transaction system in PC++ can be described by Figure ??. The four basic components are Transactions, Distributed Transaction Managers (DTMs), Local Transaction Managers (LTMs), and Atomic Objects. Each transaction is controlled by and interacts with atomic objects through a single DTM. The DTMs may simultaneously control multiple independent transactions. The DTM in charge of a transaction forwards object operations to the LTM local to the object. The LTMs are responsible for managing their own objects. Objects are responsible for completing object operations on behalf of transactions.

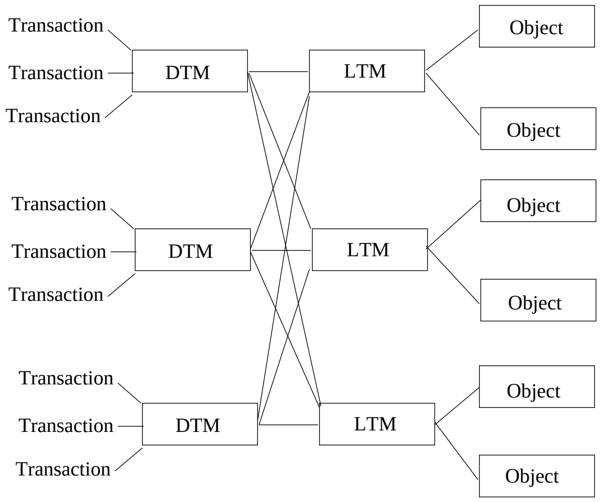

Figure 6: The architecture of the distributed transaction system

A distributed transaction T usually accesses several atomic objects resident at different sites, but it has a "home site" — the site where it originated. T submits its operations to the DTM at its home site, which is then known as the *coordinator* for that transaction. This DTM subsequently forwards the operations to the LTMs in the appropriate sites, which are then known as *participants* in the transaction. After an LTM becomes a participant in the transaction, the coordinator establishes a connection with it. Thereafter, whenever there is access to an object at that site, the coordinator forwards the operation to the participant LTM; the participant then forwards it to the appropriate object where the operation will be performed. However, when an *end_transaction* request comes from a transaction, a protocol is needed between the coordinator and participants to ensure that the commit of a transaction is atomic over all participants.

5.2 Atomic commitment

The commitment of a transaction must be atomic over all the objects involved in the transaction in order to ensure global atomicity. A transaction manager is thus introduced into the system. The transaction manager and objects cooperate according to a protocol called the *extended 2-phase-commitment* (E2PC) protocol, which is similar to the 2-phase-commitment (2PC) protocol [?] except that there is an extra layer of protocol between the LTMs and atomic objects. Like 2PC, the E2PC protocol consists of two phases. The goal of the first phase is to reach a common decision; the goal of the second phase is to implement this decision.

The E2PC protocol can be described as follows in the absence of failures:

On the distributed transaction manager side:

1. *During the read phase*: when receiving an object operation from a transaction, the coordinator forwards it to the corresponding LTM.
2. *Asking for votes*: when receiving a commit request from a transaction, the coordinator generates a timestamp for the transaction, and sends a *prepare-to-commit* command with the timestamp and the transaction identifier to each participant LTM.
3. *Making decision*: the coordinator decides to commit the transaction if all replies are "success"; otherwise, the transaction is aborted.
4. *Propagating decision*: the coordinator propagates the decision to all participant LTMs.

5. *Collecting acknowledgements*: after receiving acknowledgements from every participating LTM, the coordinator can forget all the information related to this transaction.

On the local transaction manager (LTM) side:

1. *During the read phase*: when receiving an object operation, the LTM passes it to the corresponding object.

2. *Asking for votes*: when receiving a *prepare-to-commit* command, the LTM sends an *asking-for-vote* command containing the timestamp to each participant object.

3. *Making local decision*: the LTM sends a "success" reply to the coordinator if all participant objects agree to "commit". It sends a "failure" to the coordinator otherwise.

4. *Propagating decision*: after receiving the decision about the transaction outcome, the LTM propagates the decision to every participant object.

5. *Sending acknowledgement*: the LTM sends an acknowledgement to the coordinator.

On the object side:

1. *During the read phase*: after receiving an invocation of an object operation, the object executes the operation immediately.

2. *During the validation phase*: after receiving an *asking-for-vote* command containing a timestamp for the transaction, the object validates the relevant transaction component, and returns the validation result to the LTM. If the result is "success" the object records the transaction component in its pending queue with a "waiting" status; otherwise the component is aborted.

3. *During the pending phase*: after receiving the transaction outcome, the object either aborts the component or changes the component's status to "commit" according to the outcome. Committed transaction components will leave the pending queue and enter the write phase asynchronously at each object in the order defined by their timestamps.

4. *During the write phase*: the object performs a physical validation of the component before merging any shadow copies of sub-objects created by the component with the persistent object state. If necessary, the component is re-executed before performing the merge.

5.3 Recovery

The possibility of failure always exists; recovery methods therefore must be provided so that the system can be restored to a consistent point when a failure occurs.

To ensure data consistency a system needs to provide three kinds of recovery [?]. The activity of ensuring a transaction's atomicity in the presence of *transaction aborts* is called **transaction recovery**. The activity of ensuring a transaction's atomicity in the presence of *system crashes* in which only *volatile storage* is lost is called **crash recovery**. The activity of providing a transaction's durability in the presence of *media failures* in which *nonvolatile storage* is lost is called **database recovery**.

In the DLV method, a transaction performs update operations on shadow copies of objects during its read phase. Persistent object values are not affected until the write phase of a transaction. Transaction abort is therefore achieved by simply discarding the shadow copies created for the transaction. Database recovery is independent of the concurrency control method used by a transaction system. Various methods such as *stable storage* can be used for providing database recovery.

Our crash recovery method is log-based. A *log* usually contains information for undoing or redoing all the actions that are performed by a transaction. However, in the DLV method there is no need to write a log record for every action because updates to an object can only be made on its shadow copies before a transaction commits. A log record is only needed when each phase of a transaction starts, and when a transaction completes (*aborts* or *commits*).

If a transaction was in its read phase when the system crashed, it will be aborted when the system restarts. No special recovery operation needs to be done, since the transaction neither made any change to a persistent object, nor made any promise.

At the beginning of the validation phase, each participant object in the transaction must record its *performed-operations table* (POT) and *accessed-objects table* (AOT) for the transaction on stable storage. If a transaction was in its validation phase when the system crashed, each participant object will do the validation again at restart.

If a transaction was in its pending phase when the system crashed, it will remain in this phase at restart. No special action needs to be taken. However, the *pending queue* maintained at each atomic object needs to be kept in stable storage.

The write phase of a transaction occurs asynchronously at each participant object and is separated into two or three steps: a physical validation step, possibly a re-execution step, and a merging step. A log record is necessary to indicate the end of a step.

If a transaction component was in its physical validation step when the system crashed, at restart the object will perform physical validation again for that transaction. If a transaction component was in the merging step, at restart the object will redo the merging operation. If a transaction component was in the re-execution step, at restart the object will re-execute the operations on the object.

6 Related work

Maintaining data consistency in a concurrent and unreliable environment is an important issue for distributed persistent object systems. In order to increase the level of concurrency, many researchers have suggested using the semantics of an application for concurrency control. Semantics-based concurrency control protocols can be broadly classified into two groups depending on whether they are based upon the semantics of transactions or upon the semantics of objects [?]. Lamport [?] proposed using the semantics of transactions to increase concurrency in database systems. The limitations of the traditional transaction model were demonstrated by Gray [?]. Weikum [?] introduced the multilevel transaction model in which the semantics of operations at different levels are used to increase concurrency.

However, we are interested in a more object-oriented approach which uses type-specific concurrency control by exploiting the semantics of the data through the definition of abstract data types. A locking protocol based on the notion of commutativity of operations on abstract data types was proposed by Schwarz and Spector [?]. Weihl [?] introduced the concept of atomic data types.

There are a number of systems that support transactions using atomic data types. Examples include Argus [?], Clouds [?], Arjuna [?], TABS [?], and Camelot [?]. Although they differ in detail, all of them take either an explicit or a hybrid approach.

The approach taken by PC++ is different from the other systems in the following respects. First, PC++ releases the designer from the burden of implementing the synchronisation and recovery operations by making the system do the work. Apart from specifying the conflict relation declaratively, the programmer is only required to define the behaviour of objects in a sequential and reliable environment. Second, PC++ permits the designer to represent the semantics of object operations in a declarative way instead of intertwining them with the implementation of object operations. Moreover, PC++ allows the programmer to represent enough information about the semantics to increase the amount of possible concurrency. Finally, PC++ uses an optimistic method for synchronisation while the other systems use pessimistic methods.

A proposal to add user-defined concurrency control to an object-oriented database language was made within the FIDE project [?]. Although the FIDE proposal has some similarities to the declarative approach used by PC++ to specify the concurrent behaviour of an atomic data type, it does not deal adequately with recovery. In particular, transactions are required to commit in the same order in which they were initiated, and the problem of cascading aborts is not addressed. The use of the DLV method by PC++ solves both of these problems.

The DLV method has some similarities to the method proposed by Herlihy [?]. Both methods are optimistic and both use the semantics of operations to validate interleaving of invocations by transactions. However, unlike the DLV method, Herlihy's method provides only one level of validation, and therefore always requires each component of a transaction to be re-executed at commit time.

7 Conclusions

This paper has presented a persistent object system, PC++, that is both distributed and supports semantic-based concurrency control. PC++ makes the implementation of user-defined atomic data types simple and efficient, whilst still permitting great concurrency. This has been achieved by taking an implicit approach for implementing atomic objects and allowing the concurrency semantics of object operations to be specified in a declarative way.

PC++ has already been used to re-engineer a simple distributed application, namely to maintain the database for an active badge system. Experience with this application is encouraging. The implementation is largely straightforward and the system is apparently robust. The application bears out the potential of the implicit approach for implementing atomic data types. There are many questions that require further investigation, but the general approach has been shown to be feasible.

Our future research will explore the possibility of using reflection [?] as a more flexible way of implementing atomic data types. We believe that the use of reflection as an implementation technique is well-suited to an implicit approach to defining atomic data types.

Acknowledgements

Thanks must go to members of the OPERA project at Cambridge and to our colleague at Newcastle, Brian Randell, for many discussions on all aspects of the work. Whilst at Cambridge, Zhixue Wu was supported by an ICL scholarship. He is now a Research Fellow supported by the University Research Committee of the University of Newcastle upon Tyne.

References

[1] J. E. Allchin and M. S. McKendry. Synchronization and recovery of actions. In *Proceedings of the 2nd annual ACM Symposium of Principles of Distributed Computing*, pages 31–44, August 1983.

[2] J. Bacon, K. Moody, S. Thomson, and T. D. Wilson. A multi-service storage architecture. *ACM Operating Systems Review*, 25(4):47–65, October 1991.

[3] S. Ceri and G. Pelagatti. *Distributed Databases: Principles and Systems.* McGraw-Hill, 1984.

[4] N. De Francesco, G. Vaglini, L. V. Mancini, and A. Pereira Paz. Specification of concurrency control in persistent programming languages. In *Proceedings of the Fifth International Workshop on Persistent Object Systems*, pages 126–143, 1992.

[5] J. N. Gray. The transaction concept: Virtues and limitations. In *Proceedings of the 7th International Conference on Very Large Data Bases*, pages 144–154, Sept. 1981.

[6] M. Herlihy. Apologizing versus asking permission: Optimistic concurrency control for abstract data types. *ACM Transactions on Database Systems*, 15(1):96–124, March 1990.

[7] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 6(2):213–226, June 1981.

[8] L. Lamport. Towards a theory of correctness for multi-user database systems. Technical report, Massachusetts Computer Assoc., 1976.

[9] S. L. Lo. *A Modular and Extensible Network Storage Architecture.* PhD thesis, Cambridge University Computer Laboratory, 1994. Technical Report No. 326.

[10] P. M. Schwarz and A. Z. Spector. Synchronizing shared abstract types. *ACM Transactions on Computer Systems*, 2(3):223–250, August 1984.

[11] S. K. Shrivastava, G. N. Dixon, and G. D. Parrington. An overview of the Arjuna distributed programming system. *IEEE Software*, January 1991.

[12] A. H. Skarra and S. B. Zdonik. Concurrency control and object-oriented databases. In *Object-oriented concepts, Databases and Applications*, pages 395–421, 1989.

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

[14] A. Z. Spector, R. F. Pausch, and G. Bruek. Camelot: A flexible distributed transaction processing systems. In *Proceedings of IEEE CompCon*, 1988.

[15] R. J. Stroud. Transparency and reflection in distributed systems. *ACM Operating Systems Review*, 22(2):99–103, April 1993.

[16] W. E. Weihl and B. Liskov. Implementation of resilient, atomic data types. *ACM Transactions on Programming Languages and Systems*, 7(2):244–269, April 1985.

[17] G. Weikum. Principles and realization strategies of multilevel transaction management. *ACM Transactions on Database Systems*, 16(1):132–180, March 1991.

[18] Z. Wu. *A New Approach to Implementing Atomic Data Types.* PhD thesis, Cambridge University Computer Laboratory, 1993. Technical Report No. 338.