The University Student Registration System: a Case Study in Building a High-Availability Distributed Application Using General Purpose Components

M. C. Little, S. M. Wheater, D. B. Ingham, C. R. Snow, H. Whitfield and S. K. Shrivastava

Department of Computing Science, Newcastle University,
Newcastle upon Tyne, NE1 7RU, England.

Abstract

Prior to 1994, student registration at Newcastle University involved students being registered in a single place, where they would present a form which had previously been filled in by the student and their department. After registration this information was then transferred to a computerised format. The University decided that the entire registration process was to be computerised for the Autumn of 1994, with the admission and registration being carried out at the departments of the students. Such a system has a very high availability requirement: admissions tutors and secretaries *must* be able to access and create student records (particularly at the start of a new academic year when new students arrive). The Arjuna distributed system has been under development in the Department of Computing Science for many years. Arjuna's design aims are to provide tools to assist in the construction of fault-tolerant, highly available distributed applications using atomic actions (atomic transactions) and replication. Arjuna offers the right set of facilities for this application, and its deployment would enable the University to exploit the existing campus network and workstation clusters, thereby obviating the need for any specialised fault tolerant hardware.

Key words: available systems, distributed system, fault-tolerance, atomic transactions, replication.

1. Introduction

In most British Universities, the process of registering all students as members of the institution is largely concentrated into a very short period of time. At the University of Newcastle, the registration period occupies a little over a week in September, at the start of the academic year. The purpose of the registration process is to determine which students will be taking courses within the University, and for the administration to keep its records up-to-date. From the students point of view, registration enables them to acquire the necessary authorised membership of the University, and where relevant, obtain their grant cheques. It is usually the case that students will register for particular courses, or modules, at the same time, and the information collected is used by members of the teaching staff to construct class lists, etc.

Prior to 1994, registration involved students being registered in a single place within the University, where they would present a form which had previously been filled in elsewhere by the student and their department. After registration this information was then transferred to a computer system. In 1993, the University decided that the entire student registration process was to be computerised (electronic registration) for the Autumn of 1994. The decision was also made to decentralise the registration process so that the end users of the course data, the various University departments, would have more control over the accuracy of the data entered. It was also expected that the delay before the final data could be delivered back to the departments would be considerably reduced. Although the same registration forms were issued to students, available data concerning each student had already been entered in the system database. At the registration, using unique student number as a key, student data was retrieved from the database and updated as necessary.

Needless to say that the registration process is extremely important to the University and the students: the University cannot receive payments for teaching the students, and students cannot receive

their grants or be taught until they have been registered. Thus the electronic registration system has a very high availability and consistency requirement; admissions tutors and secretaries *must* be able to access and create student records (particularly at the start of a new academic year when new students arrive). The high availability requirement implies that the computerised registration system must be able to tolerate a 'reasonable' number of machine and network related failures, and the consistency requirement implies that the integrity of stored data (student records) must be maintained in the presence of concurrent access from users and the types of failures just mentioned. It was expected that most human errors, such as incorrectly inputting data, would be detected by the system as they occurred, but some "off-line" data manipulation would be necessary for errors which had not been foreseen. Tolerance against catastrophic failures (such as complete electrical power failure, or a fire destroying much of the University infrastructure) although desirable, was not considered within the remit of the registration system.

A solution that would require the University buying and installing specialist fault-tolerant computing systems, such as Tandem [1] or Stratus [2] was not considered economically feasible. The only option worth exploring was exploiting the University's existing computing resources. Like most other universities, Newcastle has hundreds of networked computers (Unix workstations, PCs, Macs) scattered throughout the campus. A solution that could make use of these resources and achieve availability by deploying software-implemented fault-tolerance techniques certainly looked attractive.

The Arjuna distributed system [3,4,5] has been under development in the Computing Science Department at the University since 1986. The first public release of the system was made available in 1991, and since then the system has been used by a number of academic and commercial organisations as a vehicle for understanding and experimenting with software implemented fault-tolerance techniques. Arjuna provides a set of tools for the construction of fault-tolerant, distributed applications using *atomic actions* (*atomic transactions*) [6] for maintaining consistency of objects and replication of objects maintaining availability. Arjuna runs on Unix workstations, so it offered the promise of delivering the availability and consistency required by the student registration system without requiring any specialist computing equipment. In the summer of 1994 we (recklessly?) convinced the University to go electronic and committed ourselves to delivering a system that would run on a cluster of Unix workstations and provide transactional access to student records from PC and Macintosh front end machines located in various departments.

This paper describes the design and implementation of the student registration system built as an Arjuna application. The registration system been in use since October 1994, and during each five day registration period approximately 14,000 students are registered. The system illustrates that software implemented fault tolerance techniques can be deployed to build high availability distributed applications using general-purpose ('off-the-shelf') components such as Unix workstations connected by LANs. Although distributed objects, transactions and replication techniques that have been used here are well-known in the research literature, we are not aware of any other mission-critical application that routinely makes use of them over commonly available hardware/software platforms.

2. Failure Assumptions

It is assumed that the hardware components of the system are computers (nodes), connected by a communication subsystem. A node is assumed to work either as specified or simply to stop working (crash). After a crash, a node is repaired within a finite amount of time and made active again. A node may have both stable (crash-proof) and non-stable (volatile) storage or just non-stable storage. All of the data stored on volatile storage is assumed to be lost when a crash occurs; any data stored on stable storage remains unaffected by a crash.

The communication environment can be modeled as either asynchronous or synchronous. In an asynchronous environment message transmission times cannot be estimated accurately, and the underlying network may well get partitioned (e.g., due to a crash of a gateway node and/or network congestion) preventing functioning processes from communicating with each other; in such an environment, timeouts and network level 'ping' mechanisms cannot act as an accurate indication of node failures (they can only be used for *suspecting* failures). We will call such a communication environment *partitionable*. In a synchronous communication environment, functioning nodes are capable of communicating with each other, and judiciously chosen timeouts together with network level 'ping' mechanisms can act as an accurate indication of node failures. We will call such a communication environment *non-partitionable*.

The student registration system can be viewed as composed of two sub-systems: the 'Arjuna sub-system' that runs on a cluster of Unix workstations and is responsible for storing and manipulating student data using transactions, and the 'front-end' sub-system, the collection of PCs and Macs each running a menu driven graphical user interface that users employ to access student data through the Arjuna sub-system (see fig. 1).

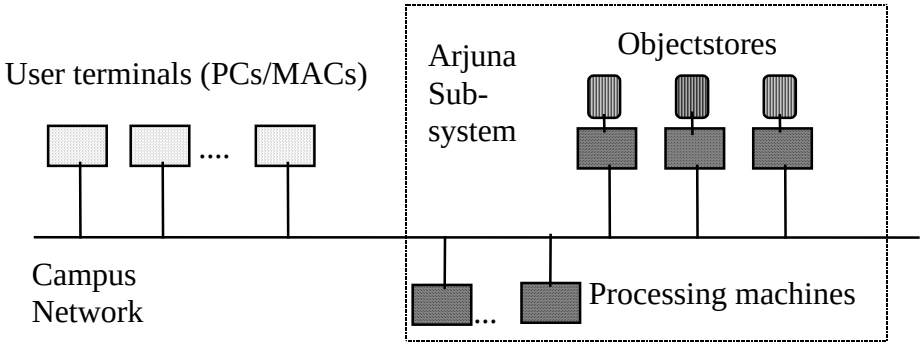


Figure 1: The student registration system.

The Arjuna-subsystem was engineered to run in a non-partitionable environment by ensuring that the entire cluster of machines was on a single, lightly loaded LAN segment; this decision was made to simplify the task of consistency management of replicated data (as can be appreciated, the problem of consistency management is quite hard in a partitionable environment). The current configuration consists of eight Unix workstations, of which three act as a triplicated database (object store). On the other hand, the front-end system was expected to run in a partitionable environment; however, we assume that a partition in a network is eventually repaired. We had no control on the placement of user machines, so we could not assume a non-partitionable environment for user machines. Note that there are no consistency problems if a front-end machine gets disconnected from the Arjuna sub-system, as the latter sub-system can abort any on-going transaction if a failure is suspected.

3. Arjuna overview

3.1. Objects and actions

Arjuna is an object-oriented programming system, implemented in C++ [3,4,5], that provides a set of tools for the construction of fault-tolerant distributed applications. Objects obtain desired properties such as concurrency control and persistence by inheriting suitable base classes. Arjuna supports the computational model of *nested atomic actions* (nested atomic transactions) controlling operations on persistent (long-lived) objects. Atomic actions guarantee consistency in the presence of failures and concurrent users, and Arjuna objects can be replicated on distinct nodes in order to obtain high availability.

The object and atomic action model provides a natural framework for designing fault-tolerant systems with persistent objects. When not in use a persistent object is assumed to be held in a *passive* state in an object store (a stable object repository) and is *activated* on demand (i.e., when an invocation is made) by loading its state and methods from the persistent object store to the volatile store. Arjuna uniquely identifies each persistent object by an instance of a *unique identifier* (*Uid*).

Each Arjuna object is an instance of some class. The class defines the set of *instance variables* each object will contain and the *methods* that determine the behaviour of the object. The operations of an object have access to the instance variables and can thus modify the internal state of that object. Arjuna objects are responsible for their own state management and concurrency control, which is based upon multiple-readers single-writer locks.

All operation invocations may be controlled by the use of atomic actions that have the well known properties of *serialisability*, *failure atomicity*, and *permanence of effect.* Furthermore, atomic actions can be nested. A commit protocol is used during the termination of an outermost atomic action (*top-level action*) to ensure that either all the objects updated within the action have their new states recorded

on stable storage (committed), or, if the atomic action aborts, no updates are recorded. Typical failures causing a computation to be aborted include node crashes and continued loss of messages caused by a network partition. It is assumed that, in the absence of failures and concurrency, the invocation of an operation produces consistent (class specific) state changes to the object. Atomic actions then ensure that only consistent state changes to objects take place despite concurrent access and any failures.

3.2. Distribution

Distributed execution in Arjuna is based upon the *client-server* model: using the remote procedure call mechanism (RPC), a client invokes operations on remote objects which are held within server processes. Distribution transparency is achieved through a stub generation tool that provides client and server stub code that hides the distributed nature of the invocation. The client stub object is the proxy of the remote object in the client's address space; it has the same operations as the remote object, each of which is responsible for invoking the corresponding operation on the server stub object, which then calls the actual object.

Arjuna creates server processes automatically on demand, as invocations to objects are made. When a client first requests an operation on a remote object, it sends a message to a special daemon process called the *manager*, requesting the creation of a new server, which is then responsible for replying to the client. All subsequent invocations from that client are then sent to this server. The created server is capable of loading the state of the object from object store where the object state resides (state loading normally happens as a side effect of the client managing to lock the object). What happens if the client does not get a response? This could be because of one of several reasons, such as: (i) it is a first invocation, and the manager is busy with other requests, so it has not yet got round to creating the server; (ii) the server has not yet finished computing; (iii) the server's machine has crashed (in which case no response will come). If no reply is forthcoming after waiting for a while, the client uses a ping mechanism to determine if the destination machine is working (see below) and retries only if the machine is determined to be working, else a fail exception is returned. Normally, client's response to this exception will be to abort the current transaction. The length of time which the client should wait is therefore crucial to the performance of the system. If the time-out interval is too short, requests will be repeated unnecessarily, but if it is too long, the client might wait a long time before realising that the server machine has crashed.

In order to better distinguish between the case where the machine has crashed and the machine is merely running slowly, Arjuna installs a dedicated daemon process on a machine, the *ping daemon*, whose sole responsibility is to respond to "are you alive" ping messages. Whenever a client has not received a response to an RPC request, it "pings" the destination machine. If ping fails to produce a response -even after several retries - then the machine is assumed to have failed and no RPC retries are made.

3.3. Object replication

A persistent object can become *unavailable* due to failures such as a crash of the object server, or network partition preventing communications between clients and the server. The *availability* of an object can be increased by replicating it on several nodes. Arjuna implements *strong consistency* which requires that the states of all replicas that are regarded as available be mutually consistent (so the persistent states of all available replicas are required to be identical). Object replicas must therefore be managed through appropriate replica-consistency protocols to ensure strong consistency. To tolerate K replica failures, in a non-partitionable network, it is necessary to maintain at least K+1 replicas of an object, whereas in a partitionable network, a minimum of 2K+1 replicas are necessary to maintain availability in the partition with access to the majority of the replicas (the object becomes unavailable in all of the other partitions) [6]. As the Arjuna sub-system was assumed to run in a non-partitionable network, K+1 replicas were required (K = 2 was considered sufficient for this particular application).

The default replication protocol in Arjuna is based upon *single-copy passive replication*: although the object's state is replicated on a number of nodes, only a single replica (the primary server) is activated, which regularly checkpoints its state to the object stores where the states are stored. This checkpointing occurs as a part of the commit processing of the application, so if the primary fails, the application must abort the affected atomic action. Restarting the action results in a new primary being activated.

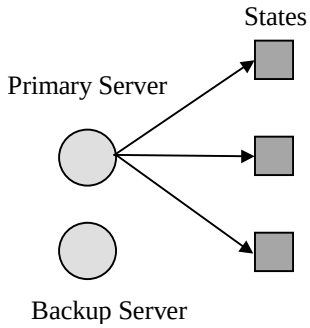


Figure 2: Passive replication.

This is illustrated in Fig. 2, where an object's state is replicated on three object stores. All clients send requests to the primary, which loads the state of the object from any one of the replicated object stores. If the state is modified it is written back to those stores when the top-level action commits. Stores where the states cannot be updated are excluded from subsequent invocations until they have been updated by a recovery mechanism. If the primary server fails then a backup server gets created. As long as a single state and server replica are available, the object can be used. Replication in Arjuna is discussed in more detail in [7,8].

4. System architecture

Based upon the experiences of the manual registration process, it was anticipated that 100 front-end machines would be necessary for the purposes of the registration exercise, resulting in a maximum of 100 simultaneous users. These machines (PC-compatible machines and Apple Macintosh systems), would be distributed throughout the University campus. For each of these two types of machine, a user-friendly interface program (*front-end*) was written, which would display the equivalent of the original paper registration form. The student data would be retrieved from an information store, written using Arjuna. In the following sections we shall examine this architecture in more detail.

4.1 The student information store

It is important that the student information is stored and manipulated in a manner which protects it from machine crashes. Furthermore, this information should be made accessible from anywhere in the campus, and kept consistent despite concurrent accesses. Therefore, a distributed information store (the *registration database*) was built using the facilities provided by Arjuna. The database represents each student record as a separate persistent object (approximately 1024 bytes), the *StudentRecord*, which is responsible for its own concurrency control, state management, and replication. This enables update operations on different student records (StudentRecord objects) to occur concurrently, improving the throughput of the system. Each StudentRecord object was manipulated within the scope of an atomic action, which was begun whenever a front-end system requested access to the student data; this registration action may modify the student record, or simply terminate without modifying the data, depending upon the front-end user's requirements.

Each StudentRecord has methods for storing and retrieving the student's information:

- *retrieveRecord*: obtain the student data record from the database, acquiring a *read* lock in the process.
- *retrieveExclusiveRecord*: obtain the student data record, acquiring a *write* (*exclusive*) lock.
- *storeRecord*: store the student data in the database; if a record already exists then this operation fails.
- *replaceRecord*: create/overwrite the student data in the database.

These methods are accessed through a server process; one server for each object.

To improve the availability of the database, it was decided to replicate each StudentRecord object, as described in Section 2. We decided to replicate the object states on three machines dedicated to this purpose (HP710s), the object stores. The system could therefore tolerate the failure of two object store machines. In addition, each primary server had two backup servers as described below.

As previously described, the registration system was expected to cope with 100 simultaneous users. Each such user has a dedicated Arjuna *client* process running on one of five HP710 Unix workstations of the Arjuna sub-system (the processing machines, see fig. 1) that is responsible for initiating

transactions on student data. Because each StudentRecord is accessed through a separate server process this requires the ability to deal with 100 simultaneous processes. The same five workstations were used for this purpose to distribute this load evenly. These machines were also used for backup StudentRecord servers; each StudentRecord object was allocated a primary server machine, with backup server machines in the event of failures. If a server machine failed, load was evenly redistributed across the remaining (backup) machines; each primary has two backups.

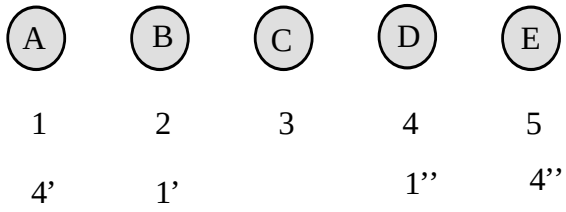


Figure 3: Server replica distribution.

Fig. 3 illustrates the server replication for 5 StudentRecord objects numbered 1 to 5. Machine A is the primary server for student number 1, with the first backup on machine B, and the final backup on D. Similarly, for student number 4, the primary server machine is D, with primary and secondary backups A and E respectively.

Each student is identified within the University, and to the database system, by a unique *student number*. With a suitable hashing function, the student number was found to provide a uniform distribution of primary servers across the available server machines. When a primary machine failure was detected, the client process recomputes the location of the new primary server for the student object based upon the new number of available machines. This mapping of student number to server location was performed dynamically while doing an 'open' on a StudentRecord.

4.2 The registration service

At the start of each registration day each front-end system is connected by a TCP connection to one of the five HP710 UNIX systems. One process for each connected front-end is created on the UNIX system; this process is responsible for interpreting the messages from the front-end and translating them into corresponding operations on the registration database. This is the Arjuna *client* process mentioned earlier, and typically existed for the day. In order to balance the load on these systems, each user was asked to connect to a particular client system. If that system was unavailable, then the user was asked to try a particular backup system from among the other machines.

Client processes, having received requests from the front-end systems, are then responsible for communicating with Arjuna server processes which represent the appropriate StudentRecord objects. As described earlier, the location of each server process was determined by the student number. If this is a request to open a student's record, then the client process starts an atomic action within which all other front-end requests on this student will occur. The server process exists for the duration of the registration action. The mapping of processes to machines is illustrated in fig. 4. In summary, the Arjuna sub-system thus consists of eight machines, of which three are used exclusively as object store machines.

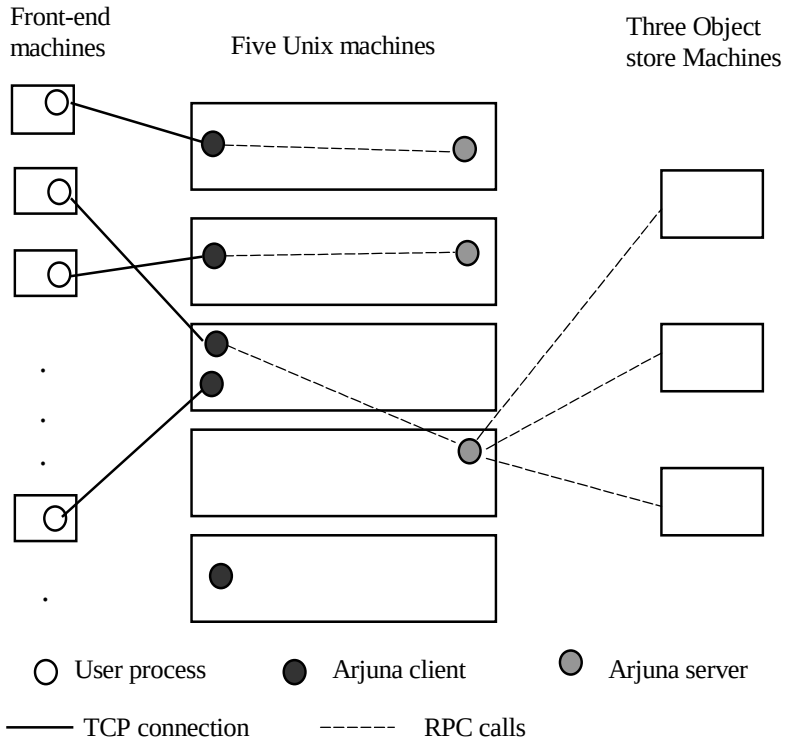


Figure 4: Client, server and user processes.

Included with the front-ends were 2-5 *swipe-stations*[1], which were introduced in the second year of operation. Registration forms were modified to include a bar-code containing the student's registration number. This was used by the swipe-stations to quickly determine the status of a student. These stations were only used to read the student's data, and therefore no modification of the data occurred.

5. Registration operation

Having described the overall system architecture we shall now examine the operation of the registration system, showing how existing students were registered, new students were added to the system, and the data was examined.

5.1 Initial set-up

Prior to the start of the registration period, the database was pre-loaded with data pertaining to existing students, and data from the national university admissions organisation who supply data concerning new students who are expected to arrive at the University. However, under some circumstances it was expected that a small number of new student records would have to be created during the registration process itself:

- A student may, for some reason, have been admitted directly by the University. This can happen in the case of students admitted very late in the admissions process. Such students do not possess a student number, and have to be assigned a valid number before registration can occur.
- There are also a number of students who, having formerly been students at the University, wish to return to take another course. It is the University's policy in these circumstances to give the student the same number as s/he used previously.

Thus, requests to create new records need to allow the user to supply the student number corresponding to the new record, or to allow the system to assign a student number.

Earlier we stated that Arjuna identifies each persistent object by an instance of a *Uid*. However, the

[1] The exact number of stations varied with the number of students.

University allocated student number has a different structure to the Arjuna Uid. Therefore, part of the database, called the *keymap*, is also responsible for mapping from student numbers to Uids. This mapping is created during the initial configuration of the database, and uses the UNIX *ndbm* database system. This mapping remains static during the registration period, and therefore each client machine has its own copy. The client is responsible for mapping from the front-end supplied student number to the corresponding Arjuna Uid in order to complete the request. The handling of requests for new records may require new records to be created, with a corresponding mapping of new Uid to student number. This will be described later.

5.2 Student record transactions

The front-end workstations run a program which presents the user with a form to be completed on behalf of a student. The initial data for this form is loaded from the registration database, if such data already exists within the system, or a new blank form is presented in the case of students not previously known to the system. The form consists of a variety of fields, some of which are editable as pure text, some of which are filled from menus, and some of which are provided purely for information and are not alterable by the user.

A registration activity consists of the following operations:

(i) either opening (asking to retrieve) the record, or creating a new record.
(ii) displaying the record on the screen of the front-end system.
(iii) either closing it unmodified, or storing the record in the database.[2]

Any such activity is executed as an atomic action. The actual operations will be described in more detail later but we present an overview here:

- *Open*: retrieves an existing record from the database. This operation is used when the record may be modified by the front-end system, and therefore a write-lock is obtained on the database object.
- *New*: for students not already registered in the database this operation allows a new record to be created and modified before being stored.
- *Close*: terminates the atomic action without modifying the record in the database.
- *Store*: stores the record in the database, and terminates the atomic action.
- *Read*: retrieves an existing record from the database, in read-only mode. This operation is typically used by the swipe-stations, and does not allow modification of the record. Therefore, the Arjuna client immediately invokes a *Close* request upon receiving the student data.

In order to start the processing of a record, the user is required to enter the student number, which is the user's method of keying into the student record database. A registration activity is started upon receipt by an Arjuna client of an *Open* or *New* request from a front-end; the client starts an atomic action and the object corresponding to that record is activated. This involves the creation of a server process, which is then requested to retrieve the object from the object store. The architecture described above clearly implies that there is one instance of a client for each active front end. Thus, there should be at most one such active object extant for each client. Although the workstation programs were intended to avoid the possibility of multiple *Open* calls being made, it was decided to insure against erroneous behaviour on the part of the front-end by implementing the client program as a simple finite state machine. Thus, following an *Open* request, further *Open* requests are regarded as inadmissible until a subsequent *Close* or *Store* operation has been performed. Similarly, *Close* and *Store* operations are regarded as invalid unless there has previously been a successful *Open* request. This is illustrated in Fig. 5.

[2] A *Delete* operation is also provided, but was disabled during the registration period.

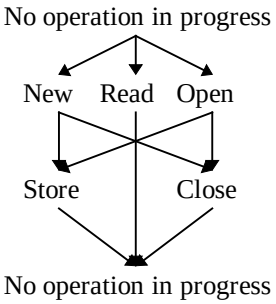


Figure 5: Possible sequences of registration operations.

The StudentRecord object is responsible for ensuring that the record is locked at the start of a transaction, and Arjuna automatically releases the lock when the transaction completes (either successfully, or as a result of some failure condition). As stated previously, *Read* operations obtain *read* locks on the student object, whereas *Open* and *New* operations obtain *write* locks.

6. The operations

In the following sections we shall examine in detail the front-end operations, and how they interact with the StudentRecord objects in terms of the protocol described previously and the operations it provides.

6.1 Open

The *Open* operation is used to retrieve a record from the database given the student number. This operation assumes that the record exists, and one of the failure messages that may be returned indicates that there is no record corresponding to the given student number. The *Open* operation first has to interrogate the *keymap* to map the student number into a corresponding Arjuna *Uid*. If an Arjuna *Uid* is successfully located, an atomic action is started, and a *retrieveExclusiveRecord* call is made to the appropriate server, chosen according to the supplied student number. The server will try to obtain a lock on the record and get the state of the StudentRecord from the object.

The *retrieveExclusiveRecord* call will either succeed, in which case the record is delivered to the front-end, or it will fail, causing an error message to be returned to the front-end and the atomic action to abort. The reasons why this call may fail are that the record does not exist, or the record is locked in a conflicting mode. If *retrieveExclusiveRecord* indicates that the record is locked, then this information is relayed directly back to the user via an appropriate message; the user then has the option of retrying, or attempting some other activity. The remaining failure is basically a time out, indicating a “no reply”. This is interpreted by the client as a failure of the server. If this occurs, then the client attempts to access the record using one of the backup servers, as described previously.

The server process created in response to the *retrieveExclusiveRecord* call remains in existence until the client informs the server that it is no longer required. This will happen either because the *retrieveExclusiveRecord* call itself fails, or because the front-end user finishes the registration activity through the *Store* or *Close* operation. The life of the server, the object, and the atomic action is precisely the duration of a registration activity.

6.2 Read

The *Read* operation is similar to *Open*, but invokes the *retrieveRecord* operation on the StudentRecord object. Because this obtains the StudentRecord data for read-only operations, such as required by the swipe-stations, the client automatically issues a *Close* request. This helps to reduce the time for which the record is locked, which could prevent other users from manipulating the data. This is the only operation which the front-end can issue which encapsulates an entire atomic action, i.e., when the student data is finally displayed the registration atomic action, student record object, and server have all been terminated.

6.3 Store

The *Store* operation is used to *commit* the atomic action and transfer the data, possibly modified by the user, into the object store database. The *Store* message generates a *replaceRecord* call to the server, which may fail because the server has crashed. This is a potentially more serious situation than if the server crashes before a *retrieveExclusiveRecord* call is made, since this represents a failure while an atomic action is in progress. All modifications made between retrieval and the attempt to save the record will be lost, but the atomic action mechanism will ensure that the original state of the record is preserved. If the *Store* fails, an appropriate message will be displayed at the front-end and the user has the option to restart the registration activity.

6.4 Close

The *Close* operation is used simply to end the registration atomic action. It is used in the situation where a user has retrieved a record, has no further use for it, but does not wish to modify it. The *Open* operation will have started a new atomic action, and have caused a server process to be created. The *Close* terminates the atomic action (causes it to *abort*) and also causes the server process to terminate. The *Close* operation cannot fail even if the server crashes; a failed server will simply impact on performance since aborting the action includes sending a message to the server asking it to terminate.

6.5 New

As mentioned previously, some students may appear at registration having no record in the database. There are two possible reasons for this, and hence two variants of the *New* operation:

(i) the student is returning unexpectedly for another year, and already has a valid student number given in a previous year.

(ii) the student is new to the University and does not have a student number. Therefore, the system allocates a new student number from a pool of "spare" numbers.

In case (ii), the pool of numbers is known before the registration begins, and blank records are pre-created and registered with the system; the mapping from Uid to student number is also known and written in the *keymap* database. In order to ensure that simultaneous *New* requests obtain different student numbers, the system uses another (replicated) Arjuna object: an *index* object, which indicates the next available student number in the free pool. This is an increment operation that atomically updates the index and returns the new number to the user.

However, in case (i) a new Arjuna object representing the student record has to be created and stored in the database, and the appropriate mapping from student number to Arjuna Uid stored in an accessible place. Because each client has its own copy of the *keymap* database, the creation of new *keymap* entries poses a problem of synchronising the updates to the individual copies of the *keymap* database on the various client machines. It was decided that the problems associated with the *New* operation could be solved by administrative action, and by accepting a single point of failure for this minor part of the operation[3]. An alternative *ndbm* database called *newkeymap* was created in a shared file store, available to each Arjuna client system via NFS. This database contained the mappings between new database object Uids and their corresponding student numbers. It was read/write accessible to users, and was protected from simultaneous conflicting operations via appropriate concurrency control.

Any *Open* request must clearly be able to observe changes made to the database as a whole, and therefore it will have to search both the *newkeymap* and the *keymap* databases. If the shared file service becomes unavailable, no new student records can be created, and neither is it possible to access those new student records which have already been created. It would be possible to minimise this difficulty by merging the *newkeymap* and *keymap* at times when the system is quiescent.

Given the new (or front-end supplied) student number, a corresponding StudentRecord object is created with a blank record. The front-end user can then proceed to input the student's information. In order to ensure that there would never be any conflict over multiple accesses to *newkeymap*, and therefore to the new number pool, it was also decided that the *New* command should be disabled on each front-end system except one, which was under the direct control of the Registrar's staff. This

[3] In excess of 12,000 students were registered over the registration periods and approximately 200 of these were added using *New*.

machine had several backups.

7. Testing and live experience

During the development of the front-end programs, tests were done to ensure that the front-end software performed satisfactorily. However, the timetable for the whole operation meant that it was impractical to mount a realistic test using the intended front-end systems themselves, involving as it would a human operator for each such station. However, it proved relatively straightforward to construct a program to run on a number of Unix workstations around the campus, which simulated the basic behaviour of the front-end systems as seen by the registration database. Each simulated front-end system would retrieve a random record, wait a short period of time to simulate the action of entering data, and then return the record to the system, possibly having modified it.

It had been estimated that over the registration period, the system would be available for some 30 hours. In this time, it was expected that of the order of 10,000 students would be registered, and that the database would need to be about 15 Mbytes in size. In some cases, the student record would need to be accessed more than once, so that it was estimated that approximately 15,000 transactions would take place. We therefore anticipated that the expected load would be of the order of 500 transactions per hour, or a little over six per workstation per hour. This however would be the average load, but it was felt that it would be more realistic to attempt to simulate the peak loading, which was estimated as follows: the human operator would be required to enter data for each student in turn; the changes to be made to each record would range from trivial to re-entering the whole record. In fact, in the case of students for whom no record was pre-loaded, it would be necessary for the whole of the students data to be entered from the workstation. It was therefore estimated that the time between retrieval and storing of the record would be between 30 seconds and 5 minutes.

7.1 Simulated operation

A program was written which began by making a TCP connection to one of the Arjuna client machines. It then selected a student number at random from the *keymap* data base, retrieved the corresponding record, waited a random period of time, and then returned the record with a request either to store or simply to close the record. This final choice was also made at random. After performing the simulated transaction a fixed number of times, the program closed the TCP connection and terminated. The program recorded each transaction as it was made, the (random) time waited, the total time taken and the number of errors observed.

The object of this test was to discover at what point the system would become overloaded, with a view to “fine-tuning” the system. At the level of the Arjuna servers, it was possible to alter a time-out/retry within the RPC mechanism (i.e., between client and server) to achieve the optimal performance, and also to tune the ping daemon described earlier.

The front-end simulation therefore arranged for the record to be retrieved, a random interval of time uniformly distributed in the range 30 to 300 seconds was allowed to elapse, and the record was then returned to the system[4]. The variable parameters of the system were:

- the range of values for the time between retrieving and storing the record.
- the probability that the returned record would have been modified.
- the number of transactions to be performed by each run of the program.

7.1.1 Results

The Arjuna ping daemon was described in Section 3.2. The failure of a machine is suspected whenever a ping daemon fails to respond (after several retries) to an “are you alive” message. In order to reduce the probability of incorrect failure suspicion due to network and machine congestion it was important to tune the timeout and retry values which the ping daemon used. By running the tests at greater than expected maximum load it was possible to tune these values to reduce the possibility of incorrect failure suspicion.

[4] With the addition of the swipe-stations, we reduced the estimated minimum time to 1 second, and tested the system with 10,000 simulated transactions in 2.5 hours.

The longest run of the test program carried out a total of 1000 simulated transactions, which took approximately 2 hours to complete. With 10 such processes running, this represented 10000 transactions in about 2 hours, or 5000 transactions per hour. This was far in excess of the expected average load during live operation, and approximately twice the expected maximum transaction rate. From these results, we tuned Arjuna to detect a crashed machine in 10 seconds. Because the simulated load was greater than that expected during registration, we were confident that we could differentiate between overloaded and crashed machines during registration.

7.2 Live operation

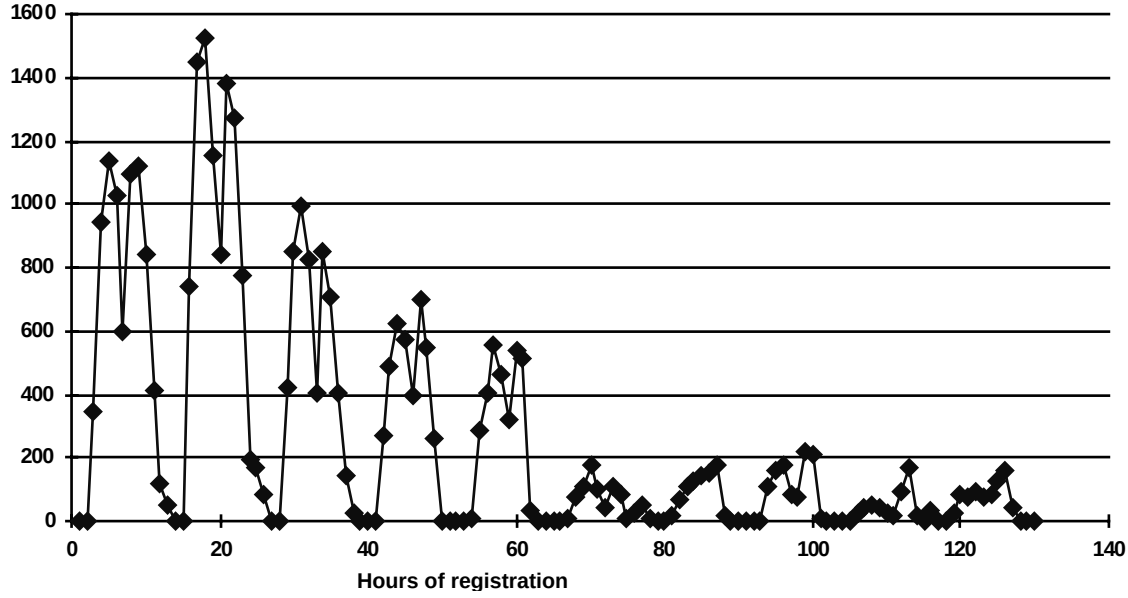
The live experience was acquired by observing, and to some extent participating in, the operation of the system during actual registration. At its peak, the system had over 100 simultaneous users performing work, and on average there were approximately 60. Many of the problems that arose during this period were unrelated to the technical aspects of the system. Such problems were: incorrect operation of the system by its operators, including attempting to repeat operations because the response was not perceived to be satisfactory, failure to install the latest version of the software on the workstations, and similar problems. In fact overall the system performed extremely well, with good performance even in the presence of failures.

There was one major difficulty that arose during the first year of operation which caused the system to be shut down prematurely (about half an hour earlier than scheduled). This occurred at the time when, and because, the system was heavily loaded, which resulted in slower than expected response times. The original figures for the number of registration users and hence the expected rate of transactions were exceeded by over 50%. Because the system had not been tuned for this configuration, server processes began to incorrectly suspect failures of object store machines. Since the suspicion of failure depends upon timeout values and the load on the object store machine, it was possible for different servers to suspect different object store machine failures. This *virtual partitioning* meant that some replica states diverged instead of all having the same states, and therefore it was possible for different users to see inconsistent states. Using timestamps associated with the object states it was possible to reconcile these inconsistencies, and the system was re-tuned to accommodate the extra load.

Although no hardware failures occurred during the first year, in the second year the registration system successfully coped with two machine failures. The machines which we were using for registration were shared resources, available for use by other members of the University. One of the Arjuna client machines had a faulty disk which caused the machine to crash when accessed over NFS. This occurred twice during the registration period when other users of the machine were running non-registration specific applications which accessed the NFS mounted disk.

7.2.1 Performance graphs

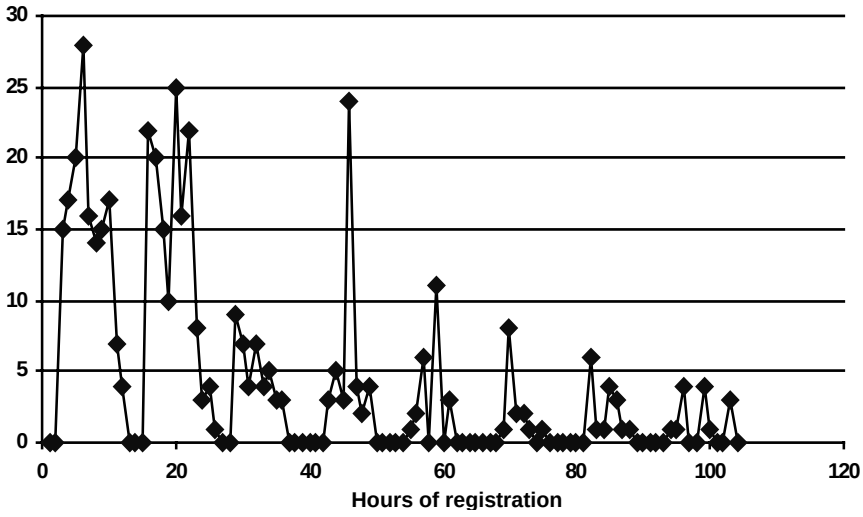
The following graphs are based upon the statistics gathered from the 1995-1996 registration period. Graph 1 shows the total number of transactions (*Open/Save*, *Read/Close*, *New/Save*) performed during each hour of the registration operation. The registration system was active for 10 days, and each working day was of 13 hrs duration; the first 5 days were the main period when students presented themselves for registration, whereas the last 5 days were used more for administration purposes. Each day is represented by two peaks, representing the main morning and afternoon sessions, with the minimum occurring when students went to lunch.



Graph 1: Number of transactions per hour.

As can be seen from the graph, the main period occurred on the second day, when approximately 10,000 transactions occurred, with an average rate of about 750 transactions per hour. The large number of transactions, and the high transaction rate can be attributed to the swipe stations, which only performed *Read/Close* operations.

Graph 1 showed all transactions which occurred during a given registration day. Graph 2 shows the number of *New/Save* operations which occurred. Most new students were registered during the first two days, with approximately 400 being registered in that period.



Graph 2: Number of requests for New records per hour.

7.3 Front-end performance

The front-end machines took some time to process a record once it had been retrieved. This was because the record itself contained much of its information in coded form, and it was thought preferable for the user to be presented with somewhat fuller information. The transformation from coded form to

"usable" form was carried out at the front-end machine. Typically, the Arjuna sub-system would respond to an *Open* request in less than 1 second, and the front-end processing would take approximately 5 seconds. Therefore, the record would be available to the user within 6 seconds of making the request.

8. Conclusions

The system described here has been used every year since 1994. The University has committed to continuing to use the registration system, and some considerable effort has gone into making the system manageable by non-Arjuna experts. The Arjuna system was used in this application to provide high reliability and availability in case of possible failure of certain components. When failures did occur, the use of atomic transactions and replication guaranteed consistency and forward progress. The system has performed well even at maximum load, and the occurrence of failures has caused minor glitches, to the extent that most users did not realise anything had happened. In the context of the ultimate purpose of the exercise, namely the completion of the registration process, the outcome was exactly what we could have wished for. This positive field experience does indicate that it is possible to build high availability distributed applications by making use of commodity components, such as networked Unix workstations and relying entirely on software implemented fault-tolerance techniques for meeting application specific availability and consistency requirements.

Acknowledgements

We thank the staff from the University's Management Information Services for willingly taking part in this 'adventure'. The work on Arjuna system has been supported over the years from research grants from EPSRC and ESPRIT.

References

[1] C. J. Dimmer, "The Tandem Non-stop System", Resilient Computing Systems, (T. Anderson , ed.), pp. 178-196, Collins, 1985

[2] D. Wilson, "The STRATUS Computer system", Resilient Computing Systems, (T. Anderson , ed.), pp. 208-231, Collins, 1985.

[3] S. K. Shrivastava, G. N. Dixon, and G. D. Parrington, "An Overview of Arjuna: A Programming System for Reliable Distributed Computing," IEEE Software, Vol. 8, No. 1, pp. 63-73, January 1991.

[4] G. D. Parrington et al, "The Design and Implementation of Arjuna", USENIX Computing Systems Journal, Vol. 8., No. 3, pp. 253-306, Summer 1995.

[5] S. K. Shrivastava, "Lessons learned from building and using the Arjuna distributed programming system," Int. Workshop on Distributed Computing Systems: Theory meets Practice, Dagsthul, September 1994, LNCS 938, Springer-Verlag, July 1995.

[6] P.A. Bernstein et al, "Concurrency Control and Recovery in Database Systems", Addison-Wesley, 1987.

[7] M. C. Little, "Object Replication in a Distributed System", PhD Thesis, University of Newcastle upon Tyne, September 1991. (ftp://arjuna.ncl.ac.uk/pub/Arjuna/Docs/Theses/TR-376-9-91_EuropeA4.tar.Z)

[8] M. C. Little and S. K. Shrivastava, "Object Replication in Arjuna", BROADCAST Project Technical Report No. 50, October 1994. (ftp://arjuna.ncl.ac.uk/pub/Arjuna/Docs/Papers/Object_Replication_in_Arjuna.ps.Z)