TCD-CS-91-33

Some Ideas on Support for Fault Tolerance in COMANDOS, an Object Oriented Distributed System

Brendan Tangney, Vinny Cahill, Chris Horn, Dominic Herity,
Alan Judge, Gradimir Starovic, Mark Sheppard

Distributed Systems Group,
Department of Computer Science,
Trinity College,
University of Dublin,
Dublin 2,
Ireland.

tangney@cs.tcd.ie

Also in ACM Operating Systems Review Vol. 25 No. 2

1 Introduction

The distributed systems group in Trinity have been concerned with fault tolerance for a number of years and are now turning our attention to the topic with renewed interest (and urgency). Specifically we are concerned to provide mechanisms for fault tolerance in the Oisin kernel [5] - the local implementation of the COMANDOS object oriented distributed system [12]. This short position paper outlines our experience gained to date, some of the lessons we have learned and the avenues which we are currently investigating in order to make Oisin reliable.[1]

2 Early Experiences and Motivation

One of our earliest attempts to provide fault tolerance is described in [16]. The approach followed was to modify the source code of an application program, running on a fault *intolerant* distributed system, in order to make it resilient in the face of multiple node failures. The application chosen was the board game Scrabble and the distributed system was an early release of V [6] running on a number of 68k processors. The algorithm for the game consists of players (which may be human or computer) and a process to manage a large lexicon structure. The players code was extended so that it:

- could **detect** the failure of other players,
- initiate, and see through to completion, the **recovery** of missing players.

The game could re-generate itself in the face of simultaneous failure of up to 3 of the 4 players![2] Further details (and there are many) can be found in [16].

A number of points are however worth noting.

[1] This document is too short to allow space to fully describe our failure model but the main assumptions are fail-stop processors, non-deterministic execution of processes and the possibility of network partition.

[2] As such it behaved like the *Worm* described in [19], more recently given a bad name by certain activity on the internet.

- The code to handle continuity took up 50% of all the application code and debugging it was **very very very difficult!**
- The mechanisms to detect and recover from failure are not specifically tied to the chosen application but have general applicability.
- While it is possible to build continuity into an application, it places a very substantial burden on the programmer. Accordingly it makes much more sense to tackle the problem of fault tolerance at the system level making it transparently available to applications! This is not to say that if applications decide to violate transparency, either to dispense with it completely or to optimize tunable parameters, that they should be prevented from so doing.

3 Operating System Support for Fault Tolerance

Oisin [5], is a kernel for distributed systems which follows the object oriented paradigm. Oisin consists of four main functional components. The Activity Manger (AM) implements the active entities (activities) in Oisin. Activities can be viewed as distributed threads of control visiting objects mapped at different nodes. Groups of related activities (Jobs) are also managed by the AM. Oisin supports the concept of (distributed) virtual object memory (VOM). When an object is being used by some activity it is mapped into VOM at some node determined by load balancing and other considerations. The distributed Storage System (SS) provides long term storage for objects. Underlying all of these components is the Communications System (CS) which provides reliable inter-node communication. Finally a full implementation of the COMANDOS kernel would include a Transaction Manger (TM) providing distributed nested transactions[3].

The plan we are following at the moment is to examine each of these components in turn and identify the *minimum* functionality that should be added to each so that the *overall system* is an integrated fault tolerant one with respect to our assumed failure model. The motivation for this overall approach is that there are a myriad of published algorithms on how to provide reliability at various levels in a system, so the challenge facing us is to implement an integrated fault tolerant environment.

Before continuing any further an obvious question must be answered. If transactions are going to be be provided why are they not used as the main building block for making the whole system reliable? The main reasons for not following this approach is that transactions on their own do not provide continuity, a recovery/restart mechanism is also needed. Furthermore the concurrency control required by the TM may be unnecessarily restrictive for non-transactional applications.

In the following sections we will describe what we believe to be the minimal support necessary from each of the functional components in order to provide an integrated fault tolerant system conforming to the COMANDOS model.

3.1 The Communications System

This is a well understood area [3]. Basically the role of the CS is to mask communications failures and to detect site failures. Currently the CS provides reliable 'point-to-point' communication with the user (sender) being notified if any fatal errors occur. The extra facilities that could be provided by the CS to help provide reliability include:

- Notification of node failure.
- Management of a site view as in [3].
- A broadcast service (c.f. [3]) providing **ordered, reliable** broadcast for use by the TM, SS, VOM, or AM.

[3] Our current implementation does not include a TM but providing one is one of our current tasks.

3.2 The Storage System

Our main requirement on the SS is that it should ensure availability of the stored data in the face of site failures and network partitions. Numerous algorithms have been published on how this can be achieved: most are based on using some form of replication (primary copy [1, 17], available copies [2], or voting [14, 20]). An interesting extra dimension to extending the functionality of the SS, could be the utilization of new storage devices, e.g. WORM disks etc., to implement **both** fault tolerance and the COMANDOS **ageing**[4] mechanism [7].

The following sections outline our current thinking on how availability may be provided in the SS.

3.3 Replication

It is well known that redundancy in the form of replication can be used to increase the reliability of a distributed storage system. Different applications may however have different requirements as regards the consistency and availability of the replicated information. For example in some cases one-copy consistency may be unnecessary[5] and availability may be more important.

One avenue being investigated is to design of an adaptive replication control method (RCM). The RCM can operate in one of two possible modes at a given time for a given object. *Consistent mode* guarantees one-copy consistency of the replicas while *weak mode* provides high availability and increased efficiency. The RCM supports controlled transition between the 2 modes. A client selects the proper level of consistency for an object by invoking the appropriate forms of the Read and Write operations.

The client is notified if the RCM cannot perform a consistent Read or Write (e.g. due to a network partitioning). If a weak Write is performed on a consistent object the RCM marks the object as potentially inconsistent. The client who invokes a consistent Read or Write on the object is informed about it's state. The client can choose the way in which the inconsistency is to be resolved. The options are:

- use the most up-to-date replica,
- use the most important replica,
- apply a given function on the value of each replica to obtain a new value,
- leave it to the client to resolve the inconsistency.

We are working on the algorithms which are to be used for the two modes of operation (candidates are: a form of voting (e.g. [15]), or a form of primary-copy algorithm (e.g. [17]).

3.3.1 Replicated Append Only Storage

In addition to the more traditional approach to replication outlined in the previous section we are also looking at the impact of changes in storage system technology on performance in general and reliability in particular.

While large, write once, storage devices have drawn some attention from the academic community for several years [18, 13, 8] they have only appeared commercially for specialized applications, rather than as a foundation for general purpose data storage. We are investigating a two tier storage model consisting of an *append only server*[6] augmented with conventional disk (and ram) client caches. This gives rise to an infinite capacity, append only, model of the storage system. The caches are **not** an integral part of the system and are only required to improve the performance of read operations.

[4] In an alternative to distributed garbage collection objects are not deleted but rather aged out to long term storage devices.

[5] This is the case when the knowledge of the semantics of the replicated information, or of the reference patterns, is available to clients to minimise the probability of the conflicting operations. In some other cases a limited inconsistency is acceptable.

[6] Based on DAT or optical disk.

A storage system build on this model is made reliable by replicating the append only storage device. In a realistic system, storage system cost will be dominated by the client caches, so this replication of the append only devices is proportionately inexpensive.

3.4 The Activity Manager and Virtual Object Memory

From the point of view of fault tolerance it is hard to separate these two components because they are so closely related. Two scenarios are outlined below, one in which reliability is provided within the AM only and the other in which VOM and AM combine to provide reliable execution.

These components taken together are responsible for masking site failure and ensuring reliable execution of jobs and activities. This by far the most difficult area, primarily because of problems with determinism. We must ensure that if failed activities are restarted they produce equivalent results as when initially run. This is particularly difficult when activities interact frequently through shared objects or make use of time or host dependent information such as the local load, time of day or some such transient value. Moreover, the picture is further complicated by the fact that activities are distributed entities which potentially span multiple nodes.

3.4.1 Reliability in the AM Only

In this scheme the burden for reliability lies with the AM. It would ensure reliability by replicating activities - one of the replicas within the *team* being designated as the *captain* and being the only one visible (in terms of its effects) to the 'outside world'. While currently there is only a single copy of an object mapped in VOM (which is shared by all activities using it) this scheme would require replicated objects in VOM. Replication, of objects and activities, has obvious disadvantages not least of which is the overhead in terms of processing resources consumed by the replicas. Ordering of invocations is difficult and there are also problems with node dependent information. Finally the system has to cope with updates to an object that is shared by two separate teams. It seems logical to put the onus on the captains who could use an ordered reliable broadcast protocol to transmit invocations on shared objects to other members of their team in a mechanism similar to that described in [11].

3.4.2 Reliability in VOM/AM

The alternative to the previous scheme is some form of of invocation logging/checkpoint scheme along the lines outlined in [4]. The state of local VOM (both memory, i.e. objects, and activities, i.e. stacks etc.) is periodically checkpointed, and a log is kept of all remote invocations. Recovery is carried out by recreating the state of VOM at some other site (possibly by reading objects from the reliable storage system) restarting the activities and replaying the invocations in some correct order (as pointed out above, maintaining the correct order is not necessarily easy). This mechanism suffers from determinism problems like the previous one (Section 3.4.1) but at least from our current perspective it appears somewhat simpler.

We have already (partially) implemented such a mechanism to ensure continuity within VOM. It is described in detail in [9] and is similar in spirit to the logging techniques described in [10] but differs in certain design issues which are not directly relevant to this discussion.

Failure detection, is achieved by sending alive messages between nodes while **Recovery** is handled by;

- maintaining a list of the objects currently mapped on each node,
- a logging mechanism that ensures all invocations on objects, in VOM, are eventually carried out.

The list of mapped objects must be maintained on nodes whose failure modes are sufficiently decoupled from the nodes they are describing.

The details of the logging and recovery algorithm are complicated but essentially involve keeping two logs per object.

- An *outgoing invocations log* of all the invocations an object makes on other objects.
- An *outgoing returns log* containing the results of all remote invocations made on the object. A *reference count* is maintained for each object so the most recent invocation can be identified. The logs are saved to the storage system whenever an object is written out.

On detecting the failure of a node the objects that were mapped there, along with the last saved logs, must be re-mapped into VOM[7]. The kernel broadcasts to other nodes looking for invocations with a higher reference count than that in the restored log. Once all outstanding invocations have been carried out, in the correct order, new ones can be accepted. Some further points are worth mentioning:

- In Oisin objects are grouped together into **clusters** so logs can be maintained on a per cluster rather than per object basis.
- The logs need not grow indefinitely but can be expunged when it is certain that an invocation will not need to be redone.
- Checkpoints can be taken to minimise the possibility of having to redo too much work. Checkpoints also allow the logs to be expunged.
- An activity's initial invocation must be stored reliably to ensure it is restarted in the event of failure.

The basic problem is to ensure that, on recovery, all invocations happen in the same, or an equivalent, order as in the initial run. While the mechanism described relies on the processing of remote invocations in order of their receive sequence number, it is not sufficient when local invocations are not handled deterministically. Thus to handle our assumed failure model this mechanism must be extended either by a form of timestamping of local invocations or by restricting the ordering of invocations with some form of concurrency control (similar to the serialisability restrictions in transactions).

3.5 The Transaction Manager

Our intention is to provide the transaction mechanism on top the underlying fault-tolerant system. The TM adds (extra) support for concurrency control (i.e. full serialisability of transactions) and the ability to abort a transaction on request or as a result of deadlock. Note however that site failures will be masked out by the VOM/AM and media failures will be masked by the SS. It will be necessary for the TM to maintain sufficient redundant information - before images or recovery points - to be able to abort a transaction. This information will be maintained at the VOM level - the SS only containing committed versions of objects.

The precise transaction system chosen may effect the choice of checkpointing system in the VOM/AM, e.g., activity based rather than node based checkpointing.

4 Summary

Traditionally, the software support for fault-tolerance has been treated as either: reliable execution of processes, or reliable storage of data. The object-oriented approach requires integration of these techniques. These ideas described here must be developed further and merged into a unified system. In particular work must be done on areas such as handling non-deterministic executions, the relationship to transactions to the rest of the system, handling network partition and interfacing to the outside world. If in the course of this effort it is discovered that the concurrency control required to ensure determinism at the VOM level is as restrictive as a transaction based mechanism then it would appear to be prudent to then use transactions as the main building block for continuity.

[7] In this work the failure model assumed that the storage system was reliable.

References

[1] P.A. Alsberg and J.D. Day. A principle for resilient sharing of distributed resources. In *Proceedings* 2^{nd} *International Conference on Software Engineering*, Los Angeles, 1976. IEEE.

[2] P.A. Bernstein and N. Goodman. An algorithm for concurrency control and recovery in replicated distributed databases. *ACM Transactions on Database Systems*, 9(4):596–615, December 1984.

[3] K.P. Birman and T.A. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, February 1987.

[4] A. Borg, W. Blau, W. Graetsch, F. Herrmann, and W. Oberle. Fault tolerance under UNIX. *ACM Transactions on Computer Systems*, 7(1):1–24, February 1989.

[5] V. J. Cahill. OISIN: the Design of a Distributed Object-Oriented Kernel for Comandos. Master's thesis, Department of Computer Science, Trinity College Dublin, Ireland, March 1988.

[6] D. Cheriton. The V Distributed System. *cacm*, 31(3):314–333, March 1988.

[7] COMANDOS Consortium. *The COMANDOS Object Oriented Architecture*, September 1987.

[8] G. Copeland. What if Mass Storage Were Free? *IEEE Computer*, 15(7):27–35, July 1982.

[9] Stephen Crane and Brendan Tangney. Failure and its Recovery in an Object-Oriented Distributed System. Technical report, Dept. of Computer Science, Trinity College, Dublin 2, Ireland., Feb 1990.

[10] W.Zwaenepoel D.B.Johnson. Sender-Based Message Logging. In *Proceedings of the* 17^{th}*International Symposium on Fault-Tolerant Computing*, pages 14–19. IEEE, July 1987.

[11] H. E. Bal et al. A Distributed Implementation of the Shared Data-Object Model. In *Distributed and Multiprocessor Systems Workshop*, pages 1–19. USENIX association, October 1989.

[12] J. Alves Marques et al. Implementing the COMANDOS Architecture. In *Proceedings of the* 5^{th} *ESPRIT Conference*, pages 1140–1157, Brussels, November 1988.

[13] J. Gait. The Optical File Cabinet: A Random-Access File System for Write-Once Optical Disks. *IEEE Computer*, pages 11–22, June 1988.

[14] D.K. Gifford. Weighted voting for replicated data. In *Proceedings of the* 7^{th} *ACM Symposium on Operating Systems Principles*, pages 150–161, 1979.

[15] M. Herlihy. A quorum-consensus replication method for abstract data types. *ACM Transactions on Computer Systems*, 4(1):32–53, 1986.

[16] P. McGrath and B. Tangney. Scrabble - a distributed application with an emphasis on continuity. *IEE Software Engineering Journal*, 5(3):160–164, May 1990.

[17] B. Oki and B. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. Technical Report Programming Methodology Group Memo 62, Laboratory for Computer Science, Massachusetts Institute of Technology, May 1988.

[18] J. Ousterhout and F. Douglis. Beating the I/O Bottleneck : A Case for Log-Structured File Systems. Internal Report UCB/CSD 88/467, University of California, Berkeley, October 1988.

[19] J. F. Shoch and J. A. Hupp. The "Worm" Programs—Early Experience with a Distributed Computation. *Communications of the ACM*, 25(3):172–180, March 1982.

[20] R.H. Thomas. A majority consensus approach to concurrency control. *ACM Transactions on Database Systems*, 4:180–209, 1979.