TMR For Off–The–Shelf Unix Systems

Eric Daniel and Gwan S. Choi
Texas A&M University, Dept. of Electrical Engineering
{edaniel, gchoi}@ee.tamu.edu

1. Introduction

This paper presents a concept study that demonstrates a transparent (non–intrusive to the application or the kernel) active replication and TMR–based fault tolerance in PC/Linux platforms. The target platform is the low–cost clusters of PCs that are increasingly more often found in the budget–oriented internet service industry. The solution we adopted is based on two ideas: 1) intercepting system calls and making the replicas agree on their parameters and return values, and 2) replacing the file system service by one that is aware of replication and can provide consistent replies to the replicas' requests. The first idea is implemented using the kernel's process–tracing hooks (`ptrace`). For the second, we modified an existing user–space NFS server. The result is a system that can preserve the consistency between replicas in some applications; although not all functionalities of Unix are currently supported we plan to extend the scope of coverage.

2. Background

The desirable factors in software–based fault–tolerance systems include portability (does a system heavily depend on a particular Unix implementation?) and transparency (does the application need to be modified in order to become fault-tolerant?). There does not exist a system that satisfies both factors. For instance, *Libckpt* [1] is a checkpointing library that can easily be linked with some applications. Although it requires very little changes to the application one wants to run, a recompilation is still necessary. Another approach is to integrate fault–tolerance into the kernel, as in [2]. This solution has the disadvantage of allowing only the applications that have been ported to that particular kernel.

3. Principle

The system demonstrated is an attempt to solve the aforementioned limitations. We cannot rely on a modified kernel, or the availability of the application's source code, but we can use UNIX's process–tracing functionalities to achieve active control of the processes. By tracing the target application we can control the interface between the application and the kernel.

This approach has some some limitations and difficulties. In particular, if we are to use active replication, we need to keep the replicas synchronized, which means eliminating the sources of non-determinism, and this is not always possible. For example, asynchronous events (signals), the use of kernel–based threads, and some other kernel effects like file locks, are difficult to reconcile with a user–space replication mechanism. Another constraint to the proposed approach is that the application software must issue a minimum number of system calls. If it spends all its time running in the user space, the error–checking mechanism will never be activated.

With these limitations in mind, we impose some restrictions on the type of application our system currently supports. The application is assumed to be deterministic when running in user space, and can communicate with its environment using 1) previously opened files (standard input, output and error), 2) regular files, and 3) certain system calls. Despite these restrictions, the system is effective for most stand–alone applications, e.g., numerical simulations.

4. System description

In this section we describe the components of our system, and how they maintain the consistency between replicas. Figure 1 shows their general organization. The replication system is composed of several elements that may all run on different machines (although it is not required):

- one *monitor*, whose role is to coordinate the other components when an application is launched, and to redirect its standard input/output towards the replicated application.
- three *replicas*: identical copies of the application process.
- one *tracer* for each replica of the application. Tracers attach to their replica with the `ptrace()` system call. They control them at the system call level, and make sure that the state of all replicas is identical.
- one *NFS server*, slightly modified to handle identical requests from several replicas.

(c)1999, Eric Daniel

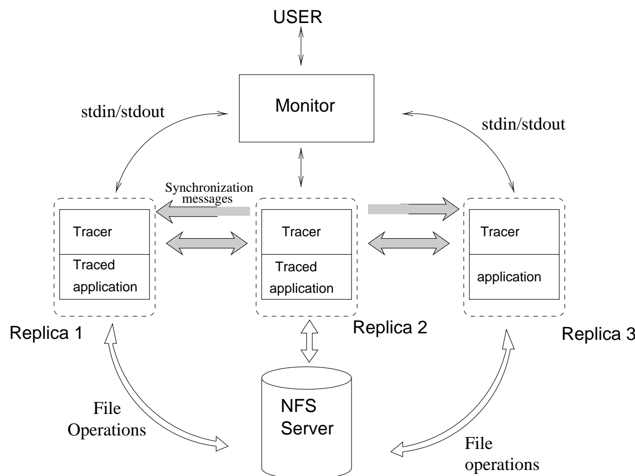

Figure 1. Architecture of our replicated environment

If a user requests a replicated process, all three tracers and the NFS server must be running. The user starts the monitor on their local machine, and indicate: 1) the hosts and port numbers of the tracers, and 2) the command to be executed. The monitor then contacts the tracers, and the application begins.

5. Fault tolerance

This section describes how the redundancy–based fault tolerance is achieved. At each system call boundary, a replica sends a message to the other replicas, and pauses until it has received the other replicas' message. The message has three functions: 1) to assert that a replica is alive; 2) to send a condensed form of the system call's parameters (a *signature*) for fault detection purposes; and 3) to help the replicas stay synchronized, by sharing informations about each other's state (like for the `time()` call).

Each replica independently checks its own signature against the signatures it received from the two other replicas. If one replica sends a faulty signature, the other replicas will notice the difference and flag the faulty replica. Flagging means that a replica will cease all communications with a faulty replica and also instruct the NFS server to do the same (to prevent a faulty replica from corrupting the file system).

Flagging also occurs if a replica fails to send its signature within a specified delay. The optimal value of this time–out delay depends on the application, and is user–definable. If this value is too small, differences in speed on the replicas may cause one replica to be mistakenly flagged. If it is too large, failure of a replica will cause a long waiting time before recovery. In our tests, timeouts of 5 to 10 seconds were acceptable.

This scheme can tolerate a single fault event (the actual failure of a replica, the corruption of a signature, or a failure notification sent erroneously to the NFS server). It cannot survive or detect two simultaneous fault events.

The network reliability is determined by the choice of the transport layer. We are currently using TCP/IP, which protects against lost packets and some transmission errors, at the cost of increased communication overhead. If desired, using multiple network interfaces and dynamic routing protects against link failures.

Although we introduced redundancy in the application, there are two remaining single–points–of–failure: the monitor and the NFS server. The monitor can be made fault–tolerant using traditional checkpointing. As for the NFS server, we can adapt an existing fault–tolerant server, FT–NFS [3]. We would need to modify FT–NFS to recognize replicated clients, the same way as we modified the standard NFS server.

6. Remarks

We have developed a software system that allows an existing program to obtain fault tolerance, without modifying or recompiling the program. This system can be easily ported to any Unix kernel that supports `ptrace()`. Single–point failures are currently addressed. For long–running reliability–critical applications, a mechanism to reintegrate a failed replica is being developed. A practical approach is to integrate a checkpointing scheme into the proposed process–replication framework, and checkpoint on a good replica only when a replica fails to generate state necessary to restore the failed replica. Efficient checkpointing algorithms for non–distributed applications exists (e.g. [1], [4]), but difficulty is implementing these algorithms in user space.

References

[1] James S. Plank, Micah Beck, Gerry Kingsley, and Kai Li, "Libckpt: Transparent checkpointing under unix," in *Conference Proceedings, Usenix Winter 1995 Technical Conference*, New Orleans, LA, Jan. 1995, pp. 213–223, Available via anonymous ftp to cs.utk.edu in pub/plank/papers/USENIX-95W.ps.Z.

[2] Anita Borg, Wolfgang Blau, Wolfgang Graetsch, Ferdinand Herrmann, and Wolfgang Oberle, "Fault tolerance under UNIX," *ACM Transactions on Computer Systems*, vol. 7, no. 1, pp. 1–24, Feb. 1989.

[3] Nadine Peyrouze and Gilles Muller, "FT-NFS: an efficient fault tolerant NFS server designed for off-the-shelf workstations," in *Proceedings of the 26th International Symposium on Fault-Tolerant Computing Systems*, 1996, pp. 64–73.

[4] Mark Russinovich and Zary Segall, "Application-transparent checkpointing in Mach 3.0/UX," in *Proceedings of the 28th Annual Hawaii International Conference on System Sciences*, Jan. 1995, pp. 114–123.

(c)1999, Eric Daniel