RFS ASSIGNMENT

A review of the Chameleon Fault Tolerance System

Wade Tregaskis

Tuesday, 9 May 2006

Cover image courtesy of Bob Wenokur of bobtography, http://www.bobtography.com/

Introduction

Chameleon is a software fault tolerance system developed by Zbigniew T. Kalabarczyk, Ravishankar K. Iyer, Saurabh Bagchi & Keith Whisnant at the Center for Reliable and High Performance Computing at the University of Illinois at Urbana-Champaign, U.S.A. Their work on the subject was sponsored in part by NASA's Jet Propulsion Laboratory, as well as Tandem Computers (now part of Hewlett-Packard).

This document reviews the system, based on the group's published paper, "Chameleon: A Software Infrastructure for Adaptive Fault Tolerance" [1]. For brevity, this will be referred to as "the paper" throughout this document.

The paper presents the design overview for Chameleon, considering its application and purpose, as well as how its practical implementation fared in real world testing. All this will be summarised and analysed in this document.

This document then considers a test case, considering the use of Chameleon for a generic web services application. In particular, what benefits Chameleon provides to the service and with what costs.

Finally, this document considers areas of Chameleon where additional work is necessary, and possible future directions.

Design Goals of Chameleon

The purposes of Chameleon is to provide an adaptive infrastructure for software fault tolerance. ‘Fault tolerance’ in this case encapsulates not only fault detection or masking, but also recovery from faults, and perhaps even adaptation around known faults.

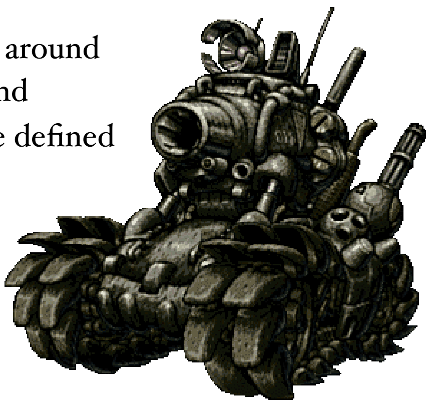
The authors note that traditionally fault tolerance has been provided by dedicated systems, whether hardware or software, for which client software must be explicitly designed and written. Their objective, in contrast, with Chameleon is to provide transparent fault tolerance for any and all off-the-shelf software. In addition, they require that this be achieved strictly in software, on commodity hardware, within a heterogeneous computing environment.

Chameleon is, as the authors acknowledge, far from the first software fault tolerance system. Many others, such as ISIS [2] and SIFT [3], are acknowledged in the paper. The authors note the distinction between these systems and Chameleon; some systems, such as ISIS, focus on distributed processing, from which fault tolerance is something of a limited side effect; others, like SIFT, provide an environment in which fault tolerant software can be written, but do nothing for software outside this encapsulation. The goal of Chameleon is to overcome these limitations, and to achieve fault tolerance specifically, not as a side effect of any other means.

Beyond the fundamental purpose of providing fault tolerance, Chameleon is intended to do so in an adaptive, extensible fashion. It does this by defining the notion of ARMORs (Adaptive, Reconfigurable, and Mobile Objects for Reliability), software units which perform some function to serve the fault tolerance system. ARMORs can be moved about the system - across computer networks - to be instantiated and [re]configured on demand, to suit the individual needs of each task. Each class of ARMOR serves a particular purpose. For example, one class of ARMOR may provide checkpointing functionality, another voting functionality, or another process execution. Linked together they provide a cohesive fault tolerance strategy, with appropriate management and autonomy. ARMORs communicate with each other over some standard network protocol, e.g. TCP/IP, via special ARMORs called Daemons.

The generic nature of ARMORs, and the flexibility with which they can be used, are intended to improve the general applicability of Chameleon by providing a general purpose architecture, with highly reusable components, but with the capability for specific tuning for each task that is performed within the system.

System Architecture of Chameleon

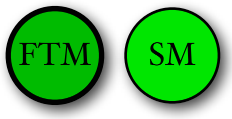


As noted, the general architecture of Chameleon revolves around basic units called ARMORs - Adaptive, Reconfigurable, and Mobile Objects for Reliability. Each ARMOR has a single defined purpose, but may be composed of multiple components, many of which are shared amongst other ARMORs - e.g. all ARMORs share a common communication infrastructure.

Multiple instances of any particular type of ARMOR may exist within the system, across multiple computers within the network. They are created on demand, compiled from source on the target host as required.

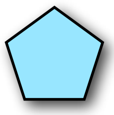
There are fundamentally three types of ARMORs, to provide the three key core competencies of the system:

1. **Managers** - these configure and operate other ARMORs, including other managers, forming a control hierarchy. They have the ability to install other ARMORs on managed computers, configure those ARMORs and then later retire them, as necessary.

   The highest level manager is the Fault Tolerance Manager (FTM), which overseas the entire Chameleon system. It interfaces primarily with lower level Managers, known as Surrogate Managers (SMs), which concern themselves with the details of a particular job's execution.

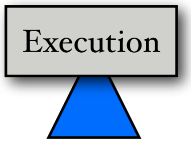
2. **Daemons** - these provide communication between ARMORs on separate machines, acting as gateways between the network and local ARMORs. They are responsible for the actual local installation and execution of ARMORs, and for monitoring those local ARMORs within the capacity of the local environment. All communication between ARMORs on separate hosts is routed via the daemon on each of those hosts.

   Daemon ARMORs also provide responses, as appropriate, to Heartbeat ARMORs, as a way of monitoring the status of the daemon, it's host, and local ARMORs.

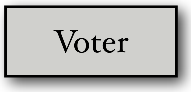
3. **Common** - a catch-all category for all other types of ARMORs, these are generally the nuts and bolts of the system; the ARMORs for performing basic tasks like executing a process, detecting process failures, voting on results in redundant executions, etc. Some key types include:

Heartbeat - This ARMOR simply monitors the responsiveness of a particular host or other ARMORs. It may do this in any number of fashions, such as a period ICMP ping to the monitored host, to ensure it has not crashed. If a monitored host or ARMOR fails to respond, the Heartbeat ARMOR notifies it's SM, which can take appropriate action.

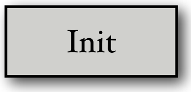


Execution - Installs & compiles (as necessary) applications on a particular host, then executes those and returns the results of that execution to it's SM. Has some rudimentary fault handling in the case of process failure, such as restarting the process.

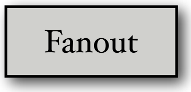
Checkpoint - Provides a facility for applications to checkpoint at defined intervals, saving their state to permit backtracking or re-execution at a later date. Interacts heavily with the Execution ARMOR, particularly when it comes time to restart from a checkpoint.



Voter - Provides a mechanism for deciding on the authoritative result of redundant executions. The mechanism may be as simple as a majority (k-of-n) consensus algorithm, looking for identical outputs, or may compute a variance between results, and permit some defined margin of variance.



Initialisation - Kind of a forward scout for new hosts that join the system. When installed and executed, it compiles a profile of the host, which can be made available to it's SM (and the FTM) and used as input for heuristics such as determining the best host on which to run a new process.



Fanout - Provides duplication of arbitrary data, communicating that data to some number of distinct ARMORs, making sure that all those ARMORs have a consistent state as regards the data - i.e. either they all receive the exact same data, or they receive nothing.

The basic premise is that some arbitrary host starts up a FTM (and daemon, and Heartbeat ARMOR), which becomes the head of the Chameleon system. Other hosts contact the FTM to join the system, installing daemon ARMORs and so forth. When a job is submitted to the system, the FTM will create a SM for that job, and the SM will then be responsible for acquiring the resources, configuring the ARMORs, and the complete execution of that job. The FTM only hears back the important data from each SM, such as final job results, or any faults that occur. In this way the FTM is only concerned with commissioning and retiring jobs and their associated SMs, not the nitty gritty of each job's operation.

As the ultimate manager of the Chameleon system, the FTM is the interface between the system and the user. As such, it will be expected to provide typical management functionality - the ability to launch and monitor jobs, to record and notify the user of failures, etc. The FTM accepts jobs described using a special semantic language, which specifies the requirements for the job, in terms of reliability, security, etc. The FTM then decides how best to meet those requirements, given it's available library of ARMORs, the network environment at the time, and the various Fault Tolerant Execution Strategies (FTES) it is aware of. Presumably these FTESs are stored in an easily modified form such that they can be added to as new strategies are devised.

The FTM is also a critical point of failure, and consequently a backup FTM is also initialised alongside the primary FTM. All ARMORs send data to both the primary and the backup FTM. If an ARMOR detects a problem with the primary FTM, it can notify the backup FTM. The backup FTM can then decide, if necessary, to forcefully retired the primary FTM, promoting itself to the position. Another backup FTM is then configured to monitor the new primary FTM.

The idea behind the generalised architecture, with numerous ARMORs working together, is to provide flexibility. Each job submitted into the system can have different requirements[1]. For example, one particular job may require high reliability; that is, that its results are correct. The FTM may thus decided to use Triple Modular Redundancy (TMR) as its FTES. Thus, it creates a new SM and tells that SM the FTES being used (TMR). The SM then creates three Execution ARMORs and a Voter ARMOR, as dictated by the FTES. The resulting configuration is as shown in figure 1.

When the Voter has all the results, it can determine if there is a majority consensus (for example), and communicate the result to it's SM. The SM may then communicate a successful result back to the FTM. Or, perhaps the Voter ARMOR is configured such that if the there is not unanimous consensus amongst the three executions, it will perform additional executions (via the SM) to reach a higher certainty (e.g. 90%, requiring at least nine identical results for every divergent result).

1 Indeed, particular tasks within a single job can have completely different requirements.

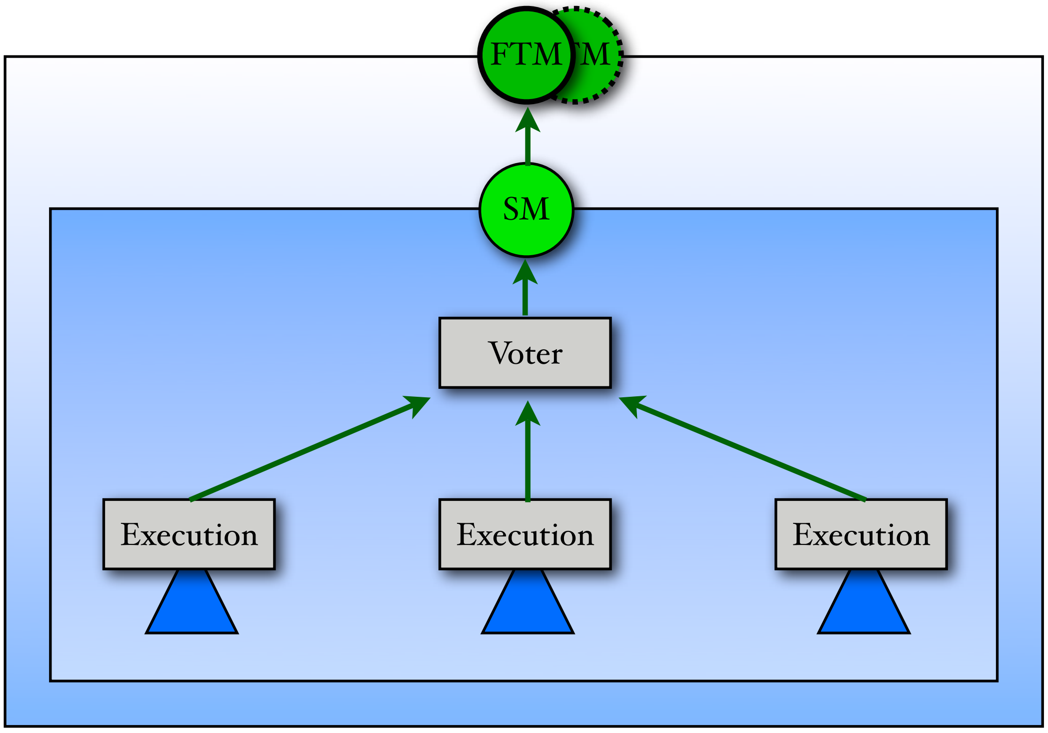


Figure 1 - Example TMR Configuration

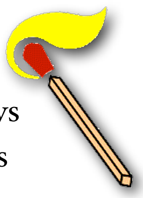
In the above example, the system provides TMR (Triple Modular Redundancy), with the aim of handling faults by masking them. This is but one possible FTES. As another example, consider another job which simply needs to be run through to completion, without strict reliability requirements, but which takes a very long time to complete. TMR is likely not the best FTES for this job. Instead, an FTES involving checkpointing is more suitable. This job could be run by an Execution ARMOR coupled with a Checkpoint ARMOR; periodically the Checkpoint ARMOR will save the state of the job, so that in the event of an error the job can be rolled back to the most recent checkpoint. In this case, the primary focus is on managed restart in the event of a failure, not fault masking. This FTES does not offer the same assurances of reliability as the TMR FTES, but may be a better trade-off given the significant resources required for this long-running job.

There are numerous other cases that could be considered, but that concludes a suitable introduction to the system. The point to be taken away from these examples is that the system can provide a wide range of strategies for detecting and handling faults, as suited to each particular job, or indeed each sub-task of a job.

Fault Tolerance Strategies Involved

The Chameleon approach is quite adaptive, and as such the overall FTES used for any particular task is typically a hybrid of several different fundamental FTES. Nonetheless, there are clear strategies for which each ARMOR is tailored.

It is important at this point to differentiate between faults, errors and failures. Using the definition provided by Ben Soh in the RFS lecture notes [4], the three terms are defined as follows:



- **Faults** are an undesirable occurrence, such as a random hardware glitch, loss of network connectivity, etc. They may be transient (the fault occurs and then disappears without need for intervention), intermittent (the fault is always present, but does not always induce an error), or permanent (the fault is always present and always induces an error).

- **Errors** are the incapability of the system to correctly handle a fault - such as a process assuming it has access to a particular file or network port, when it is not always the case, or not being able to handle improperly formatted input. Errors are always present in a system - they are a logical failure of the system. They only cause failures when a fault occurs which invokes the erroneous behaviour.

- **Failures** are an instance in which a fault occurs, and because of an error leads to incorrect results. This may be a simple outright failure, such as an application crashing or locking up, or a more subtle corruption of output, such that it is inaccurate, incorrect, or even outright garbage. Catastrophic failures are those which lead to serious damage - e.g. injury, loss of life, or other such major consequences.

Throughout this document "fault" is typically used in a more specific sense, meaning any fault which, through an error, leads to failure. Faults which never lead to failures are not a focus of Chameleon.

Chameleon generally treats processes as black boxes. When failure is discussed in this document, it refers to failure of a process, as seen by an external viewer (i.e. Chameleon). Failures which occur inside a process and are recovered from inside that process are not of concern to Chameleon; it is certainly possible and always encouraged for individual processes and applications to provide as much of their own fault tolerance as possible, in addition to whatever abilities are conferred to them from systems like Chameleon.

Fault/Error Detection

The first, critical step in a fault tolerant system is the acceptance that faults will occur, and thus there must be mechanisms for detecting them. Although it is common to talk about "fault" or "error" detection, such things are rarely detected first hand. Instead, what is typically watched for are failures. Failures, from Chameleon's point of view, are nearly always detectable by watching an application's state and I/O, since by definition failures must have an effect on the application's function, as externally visible. Since failures are relatively easy to monitor, the errors that cause them can be inferred. From these, likewise, can be inferred, to a degree, what fault occurred.

On the other hand, faults which do not trigger errors (and subsequent failures) are rarely detected. However, this is generally okay - if a failure does not result, then the fault is likely not of concern. For example, if a network link is temporary lost - a fault - but the program detects this and uses a backup link, then no error is present, ergo no failure results, and Chameleon need not be overly concerned[2]. Conversely, if the program instead aborts when it cannot use the link, that is an error, which Chameleon handles. Nonetheless, it is useful to detect all faults, whether or not they trigger errors or cause failures, as a high rate of faults may indicate a higher likelihood of encountering an error, even if it has not yet. While Chameleon does not have provisions, as presented in the paper, for predicting failures, it is certainly something that could be added[3].

Detecting failures is fundamental and critical, as the end user needs to know that the results they ultimately receive came from a successful run[4] of their application; if failures pass through undetected, the results are not trustworthy.

Failures are divided into three domains in the paper (which refers to them as errors, although this is not consistent with the terminology used in this document):

1. **Abnormal termination** - where an application exits unexpectedly, whether by crashing or voluntarily on internal detection of an error.
2. **Value-domain** - where the output of an application is incorrect, due to, for example, a hardware fault leading to corrupt data or incorrect control flow.
3. **Time-domain** - where an application ceases to make progress in it's task, becoming "live-locked", and requiring external termination and recovery.

While failures in the Abnormal Termination domain are relatively trivial to detect, those in the other domains are not always so easy to catch. The difficulty in detecting value or time-domain failures is that many do not visibly manifest themselves outside the

[2] Excepting, of course, when considering efficiency, but that is for a separate discussion, included later in this document.

[3] Indeed, the paper acknowledges the usefulness of such functionality, as part of the larger issue of resource management.

[4] Note the difference between a successful run of an application, and correct results. A successful run indicates no errors occurred ***within the application***; if there was an error in the input to start with, then of course the output may still be incorrect.

application. Logic (software) errors in particular may not cause the application to crash, but may nonetheless render the application's output incorrect. For a general purpose system like Chameleon, without expert knowledge on each application, this poses a problem.

For faults which are transient or intermittent, each execution of the application may produce different results. Thus, FTES of multiple execution (e.g. dual-execution, TMR, etc) - whether spatial or temporal - can detect these faults (and possibly mask them). The Voter ARMOR is used extensively in these cases.

If, on the other hand, the faults are permanent - given a particular environment at least - FTESs such as TMR will not necessarily discover them. There may be some way to infer the presence of such a fault by analysing the output in context[5], but this is not something the Chameleon can do with it's generic ARMORs[6]. What Chameleon can do is run applications redundantly in different environments, to try and weed out any faults or errors particular to an environment. In can do this, for example, by using TMR with each of the three executions run on different architectures.

Other faults may manifest as application crashes (or worse, host crashes). These are trivial to detect - in the case of the application itself crashing, the Execution ARMOR will be notified by the host operating system when the application exits, and can check it's exit status and outputs. The SM can then act appropriately to reinstate the application.

If a host crashes, there is no direct notification mechanism. Thus the reason for the existence of the Heartbeat ARMOR, to continually poll a given host, and notify the FTM if it does not respond for an extended period. The FTM can then, at it's discretion, initiate recovery of the unresponsive host's ARMORs and applications on another host.

Chameleon can also detect live-lock in at least some cases. The Voter & Execution ARMORs both include functionality for detecting live-lock, by essentially benchmarking a given application or ARMOR with a short sample task, then using that to gauge how long the real task should take. Unfortunately, this relies on being able to determine the amount of time required for a given task, which may not be predictable. Thus, live-lock detection is not always reliable. It can be aided by consideration of user parameters on the application; by manually specifying a maximum duration for the application's execution. It can also be negated, to a degree, by regular checkpointing - thus shorter

[5] This comes into the domain of expert software, a trivial example of which is knowing that the output for a given process should be a alphanumeric, comma-separated list, and thus knowing when it is instead outputs binary blobs that something has gone wrong.

[6] Although, of course, the extensibility of Chameleon certainly permits the addition of "expert" ARMORs, which could perform application-specific validation.

time-outs can be used, as recovery need only go back to the latest checkpoint, not start execution again from scratch.

Nonetheless, live-lock detection is something the paper seems to brush under the rug, as it were, without presenting a satisfactory solution to. It is certainly a difficult problem, one which may not have a perfect solution.

Beyond these faults & errors in the applications themselves, there may be faults & errors within the host or network. For example, for a distributed application - one that executes in separate processes, possibly on separate hosts, with communication between them - faults may occur that cause corruption of communications. In an advanced Chameleon implementation, this type of failure can be detected; a to-be-designed ARMOR can be installed to watch the communications of each process, to ensure that all processes see the same I/O between them - i.e. that communications are not lost, corrupted or improperly duplicated.

Such an implementation, however, requires hooks into the operating system on the application's host in order to monitor it's I/O. Such functionality is not usually provided easily; typically it requires super-user privileges at the least. As such, while it is noted in the paper, it is not presented as a key feature of Chameleon.

Alternatively, Chameleon can provide an API to the distributed application to allow it to work with Chameleon in performing tasks such as checkpointing. Alas, this is of course only applicable for specially developed applications.

While it's been shown thus far how faults are detected in application software, how are faults in the Chameleon system - it's component ARMORs and network - detected? In short, there is minimal provision for this. ARMOR crashes are detected by whatever ARMOR is upstream of the failure (typically a daemon or manager). But for failures other than outright crashes, there is little protection. While each ARMOR will perform basic validation of data it receives from other ARMORs, there is little redundancy in the Chameleon network itself, aside from the backup FTM and the general flexibility in the system for restarting lost or broken ARMORs. The paper notes this deficiency, but does not present any significant solution. It can be immediately seen that simply applying the system to itself does not solve the problem, merely make it recursive and ultimately worse (more unreliable components simply increases the probability of failure). Algorithmic changes are ultimately necessary to improve Chameleon's own fault tolerance, which is discussed later in this document, in the section on Future Work.

Fault/Error Isolation/Containment

Once a fault or error has been detected, the first key action is to contain the fault. This primarily means trying the following, in order of preference:

1. Preventing the fault triggering an error.

2. Preventing the error causing failure.
3. Preventing failure from propagating.

Chameleon is primarily concerned with the 3rd stage; the majority of errors in the first stage are in the applications Chameleon executes, or in the hardware it operates on, and both are beyond Chameleon's abilities to directly fix. Chameleon can interject for some errors, to prevent failures - e.g. ensuring reliable communications between processes - but again is largely limited by forces outside it's control.

In order to ensure the safety of the entire system, all failures must be either properly handled (e.g. masking), or else the task cancelled and the user uniformed of the failure. Strategies such as TMR provide the ability to mask some faults and errors, which is a reasonable approach, but only for failures rare enough as to be unlikely to effect the result of the vote (i.e. highly unlikely to occur in two of three parallel executions). And this only covers failures in application output; for faults that cause the application to crash or live-lock (stop making any more progress), redundant executions may not be the solution.

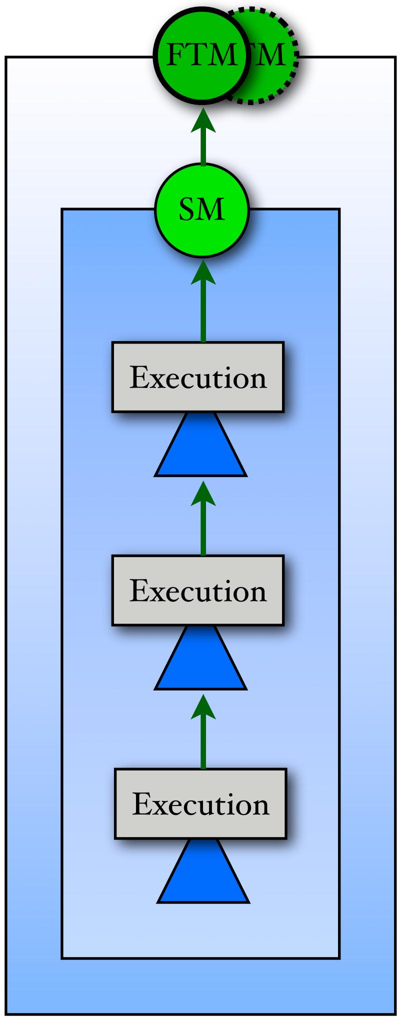


Figure 2 - A multi-stage application

For faults which lead to incorrect outputs, it is imperative that the incorrect data does not propagate, whether back to the user or just to the next stage in the job. Consider an example case, where a series of processes are executed in series, each taking the previous processes' outputs as it's input. If this were executed naively, as shown in figure 2 (previous page), an error in the data at any point would continue through the system, most likely producing incorrect data at the end.

If this entire system is run in parallel, say in triplicate - as shown in figure 3, below - then reliability is improved somewhat - the odds of the same failure occurring in two parallel paths is quite small. Thus, the system is reasonably safe. However, the odds of two parallel paths having failures of any kind is still quite large. This will prevent the Voter from reaching a conclusion, and reduces the reliability of the system.

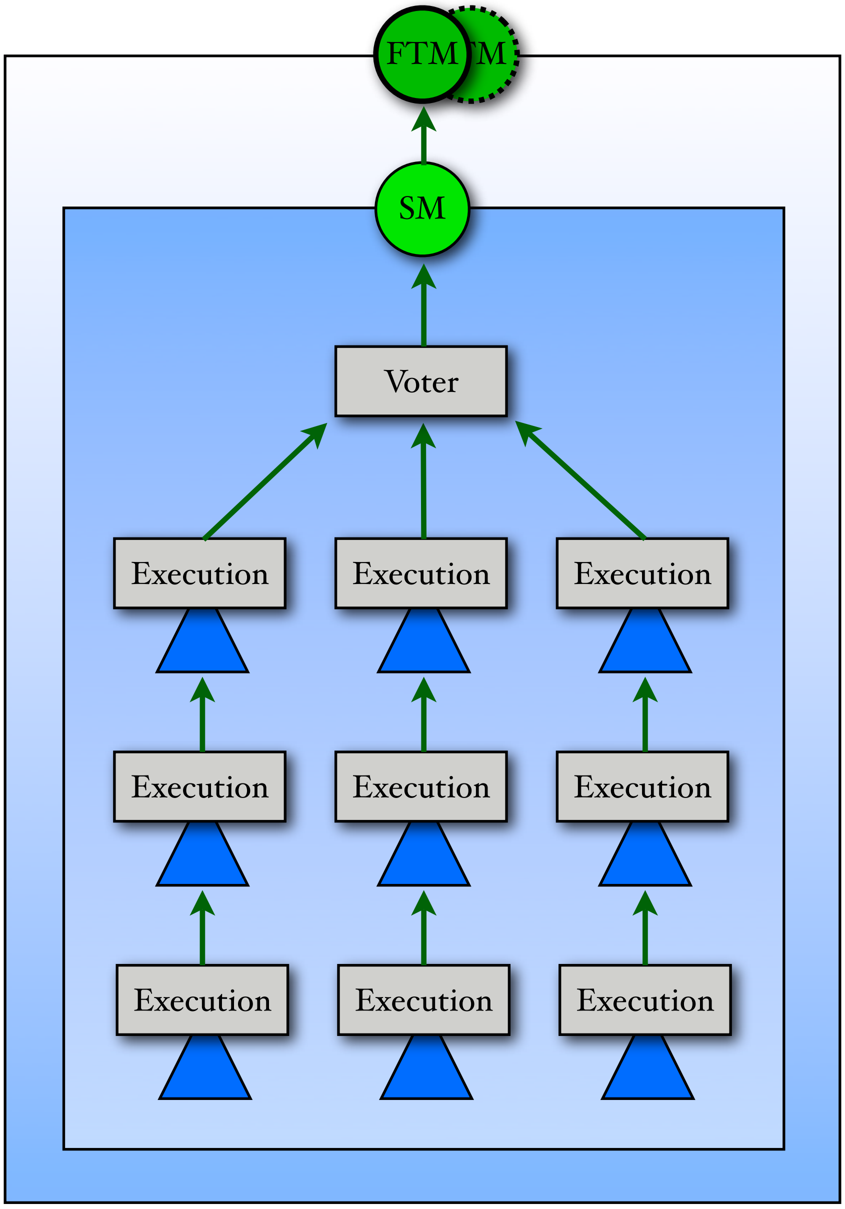


Figure 3 - Naive TMR implementation for a multi-stage application.

Note, however, that the probability of failure over all three stages is the sum of the probability of failure at each stage. Thus, if failure is detected at each stage (and corrected), then probability of an overall failure is greatly reduced, and the reliability and safety of the system is increased.

Thus, Chameleon would prefer to execute each individual stage separately, with a Voter ARMOR choosing the master results at each stage - see figure 4 (next page). The Voter can then pass the chosen results to a Fanout ARMOR, which can distribute the exact same results to each parallel copy of the next process.

Other than this redundant execution approach, Chameleon does not really provide any other mechanisms for containing I/O faults & errors. In particular, as previously noted, without special OS-dependant extensions to monitor all I/O, Chameleon cannot provide acceptable levels of containment for processes which communicate amongst each other outside of Chameleon. As such, it is not easily applied to distributed applications. Given that Chameleon's favoured FTES involve redundant execution, which distributed applications typically provide anyway, this may not be as much a loss as it first appears.

A final note on possible problems - while the Checkpoint ARMOR can rollback to the previous checkpoint once an error is detected, there is no guarantee the error was not already manifest in the checkpoint, and may simply recur. This scenario is not detailed in the paper; presumably the Checkpoint ARMOR could progressively roll back through checkpoints until the error no longer recurs, but no exact mechanism for this is presented. The Checkpoint ARMOR is still covered by the Execution ARMOR, so if nothing else after a certain number of failures the Execution ARMOR will terminate the local process and defer to the SM, which may try to reconfigure the system around the problem.

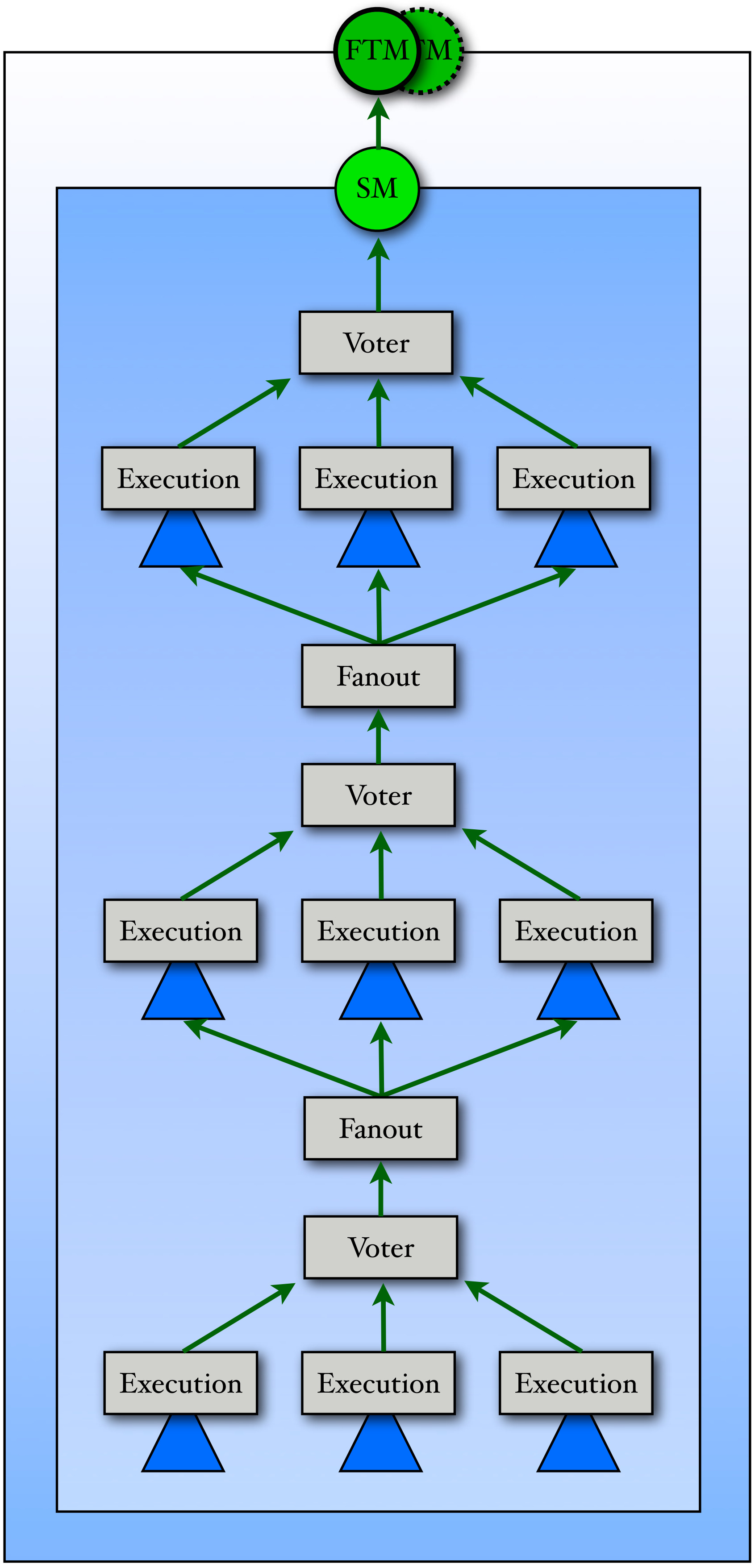


Figure 4 - Improved TMR implementation for a multi-stage application

SYSTEM RECONFIGURATION

In the case of some types of failures, such as a process crashing, merely trying again is not always a solution; the failure may recur indefinitely. Consequently, Chameleon is designed to reconfigure itself in response to repeated failures.

To start with, the simplest response to a process failure is to restart the process. This is the Execution ARMOR's first reaction to such an event. For transient or even intermittent faults, this may be acceptable - sooner or later the application will conclude successfully. Chameleon goes further than this, however - if an application fails repeatedly, the Execution ARMOR defers to its SM, which may decide to move the application to another host. After further failures, it may even try a different platform.

For example, if a process fails on a given host multiple times, the responsible Execution ARMOR notifies its SM. The SM may decide to move the process, along with its Execution ARMOR, to a different host, and try again. This provides responsive temporal redundancy. All the while, other fault tolerance mechanisms may be in play - for example, above all this there may be a Voter ARMOR which takes the final output, once the crashing has been resolved. See figure 5. The Voter ARMOR need not know that any of it's sources had to relaunch their process, or were moved to another host.

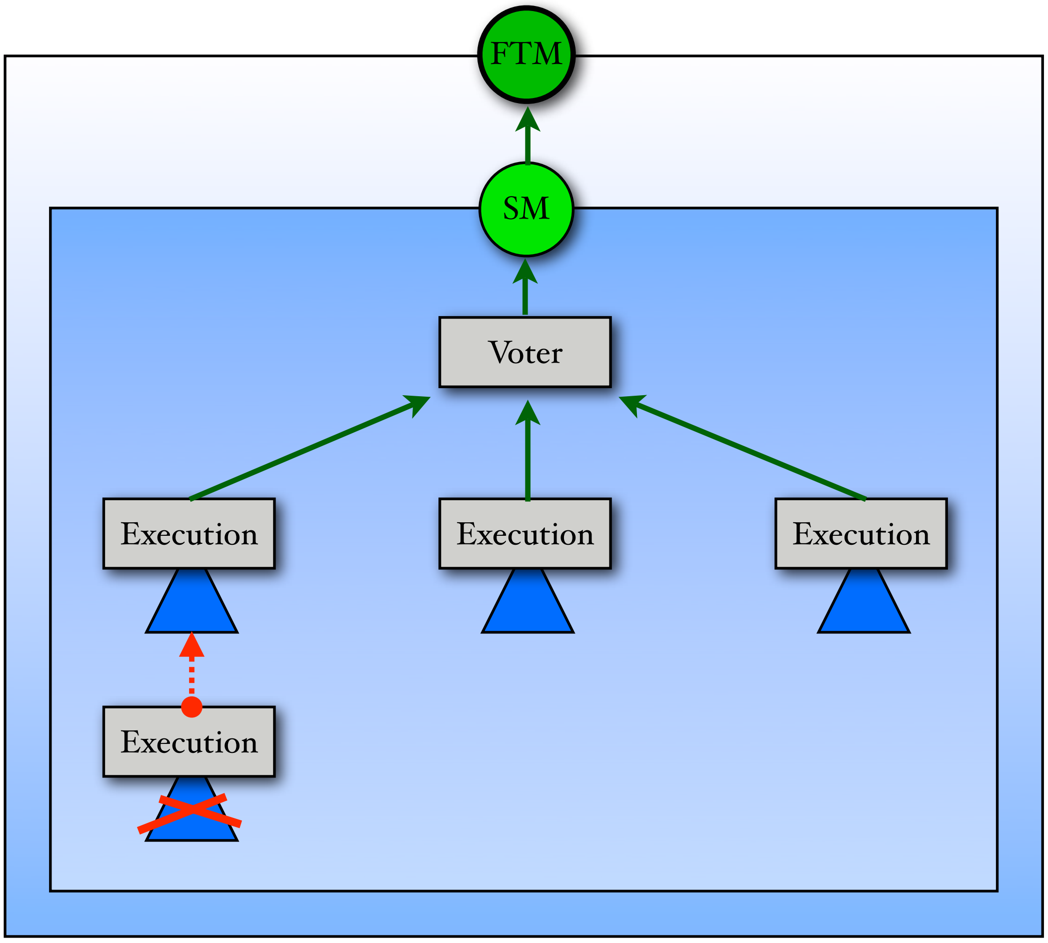


Figure 5 - Black box behaviour of the Execution ARMOR

An emphasis is placed on retrying repeatedly failing processes on different architectures, to remove from the equation as many environmental factors as possible. A still failing process is eventually just aborted, with the user notified of the problem. The assumption at this point becomes that the process has a bug in it which the user must resolve. In this case it is beyond the capabilities of Chameleon to fix, so the task terminates, but clearly as a failure, so that the system's safety is ensured.

System Recovery

It has already been mentioned several times that failure of a process leads to its execution being reattempted. Further failures may prompt the SM responsible to reconfigure the processes' environment, or try a variation of the desired FTES, in an attempt to coax away configuration-specific faults and errors. If this still fails, the SM aborts the job, and the FTM conveys the inability to perform the job back to the user.

If an ARMOR fails, once that failure has been detected the FTM can create a replacement, which can then step in and more or less pick up where the previous instance left off. Chameleon requires all ARMORs to remain available for as long as they may be needed, even for the most pathological scenarios. For example, Execution ARMORs should persist even after their process has finished, until the whole job has concluded. Consider, for example, a standard TMR setup. If the Voter ARMOR fails before the job has concluded, it must be replaced. But the replacement will need all the output from the Execution ARMORs. The Execution ARMORs do not throw this output away until they themselves are retired. Thus, they are able to resend it to the new Voter, and the system still performs as desired.

If a daemon ARMOR fails, the FTM can immediately move effected ARMORs to a different host, restarting them there as best it can. In the meantime, it can also try to re-establish a working daemon ARMOR on the effected host. "Daemon failure" can encompass the daemon actually failing, the host failing, or a communications error. Thus, the FTM would not wait on the host to return; but it would make all attempts to regain contact with or reinitialise the host.

If a SM fails, it can be restarted by the FTM, and the work done by the other ARMORs should not be lost; the FTM will notify them of their new SM, and they will resume or restart their communications with the SM as appropriate. Nonetheless, the loss of a SM is relatively expensive, as for complex FTESs it can entail a lot of repetition, with the latency and bandwidth use that entails.

If the primary FTM fails, the backup FTM should be able to replace it with relatively little disruption. Since Chameleon tries at all times to maintain consistent state between the primary and backup FTMs, the transition should be relatively quick. There may nonetheless be delays as the backup FTM asserts control over all the ARMORs in the system, and reverses any damage that may have been done by the previous FTM.

Chameleon System Performance Evaluation

To evaluate the performance of Chameleon, we must first define the metrics that are valued in a fault tolerant system. Ultimately, what we care about is dependability - the assurance that the system will operate correctly. Dependability is a function of several key metrics, such as reliability, availability, safety, security and responsiveness. It is also useful to consider things such as fault coverage and efficiency.

Reliability

★★★★☆

Reliability is measured as the probability of failure within a given time frame. Working on the assumption that perfect reliability is impossible, this ultimately boils down to how much users can trust the results of an application executed within Chameleon. Given that there is a practical limit to how many faults can be removed or prevent, to have high reliability the system needs to be sure that most remaining faults are detected and corrected. "Most faults" means most of those which in practice actually *occur*, not necessarily those which *may* occur; that becomes an issue of fault coverage, discussed later.

Thus, what must be considered is what faults are most likely. This is highly dependent on the exact environment Chameleon operates in, and what sorts of applications it manages. Generally, the most common faults are abnormal termination and connectivity failures. Chameleon handles these with ease, as has already been discussed and demonstrated.

Less likely, but still significant faults include whole hosts failure and process live-lock. The former is easily handled by Chameleon, with some finite delay, while the latter is not so clear. As noted, live-lock can be difficult to detect, but with the provision of special APIs to applications for watchdog monitoring, the odds of a false negative can be significantly reduced.

The least likely events are those that are largely random - such as an electrical glitch in a CPU or memory, causing data corruption or an unexpected change of control flow. These faults can be detected quite trivially using redundant execution, which Chameleon employees widely.

In summary, the Chameleon system, while of course not perfect, does appear to cover the vast majority of important cases, and as such can be considered successful in this regard.

Availability

★★★⯪☆

Beyond just being reliable, a system also needs to be available - that is, it must be operating correctly and ready for use. Given finite resources with which to perform it's tasks, Chameleon must use them wisely in order to ensure they are available as necessary. Furthermore, it must be able to recover from faults quickly.

Because of Chameleon's dynamic nature - being able to move ARMORs and applications from host to host - and it's high tolerance to localised failures - such as a host or ARMOR failure - it is able to provide high availability, within those resource limits. In particular, it has the capacity to do load distribution (this was noted, although not implemented, in the paper).

Some of this availability is nonetheless contingent on the size of the Chameleon network. A network with a hundred Chameleon hosts will obviously offer better general availability than a network with just five.

The recovery time for many faults can be significant. A host becoming unavailable, for example, can take 10 seconds or more to detect. There's then a few seconds required to restart the effected ARMORs on other hosts. All the while, for the user(s) who's tasks were on the unresponsive host, the system is effectively unavailable. Thus, in this respect Chameleon leaves something to be desired.

Safety

★★★★☆

A safe system is one which is either working correctly, or which has detected a fault and avoided catastrophic failure. For all the faults that Chameleon detects, it is safe - if it cannot mask or otherwise avoid the fault after some effort, it will notify the user of the problem. It will not, for example, present the results of a process which keeps producing invalid output. In this regard, it is safe. For faults which it does not handle, it may not be safe. This then becomes an issue of fault coverage, discussed later.

Security

☆☆☆☆☆

This encompasses not only privacy of data, but also it's authenticity & integrity, and the system's general resistance to malicious behaviour. This is one area in which Chameleon is extremely lacking. There are no mentions of encryption in the paper, nor any notes on the huge danger presented by the practice of compiling and executing arbitrary code on a host.

Additionally, the paper does not consider at all the possibility of malicious ARMORs in the system, intentionally subverting the system. For example, a malicious Voter could return aberrant results, instead of the majority consensus. Any of the Voter, Execution and Fanout ARMORs could modify data at their discretion, with no recourse for detection by the FTM and thus no way of informing the user of the violation.

There is mention of using CRCs and checksums, which provide some defence against random faults, but do absolutely nothing for security.

A system that is secure against malicious behaviour is consequently highly secure against random behaviour. Thus, Chameleon's lack of security is a significant concern.

RESPONSIVENESS

★★★★☆

Chameleon, in the incarnation presented in the paper, is not positioned as a real-time system, although the paper does devote some space to discussing how it could be made into one. Fault tolerant real-time systems are exceedingly difficult to create, because they rule out many of the temporal redundancies that can otherwise be employed. Real time fault tolerant systems typically employ spatial redundancy and active backups; Chameleon offers this functionality, but not explicitly designed with time-critical usage in mind.

Nonetheless, responsiveness doesn't require being hard real-time. In most cases, the responsiveness that is important is how long it takes to complete a task, in wall time. Chameleon appears to favour spatial redundancy for this reason - while it uses more resources at a time, it provides more or less the same responsiveness as a single execution, with the benefits of redundancy for fault detection and masking. Chameleon also uses temporal redundancy, although only where it is unavoidable or where responsiveness is not important.

The paper includes benchmarks of Chameleon's responsiveness in the test network used by the paper's authors. They show that it takes in the order of one second to install an Execution ARMOR, for example, which is insignificant compared to the typical task duration. For faults leading to application crashes, Chameleon took on average just under one second to respond to the failure. For host crashes, Chameleon takes a little over 10 seconds - which is limited by the configurable timeout; 10 seconds in this case.

Once faults were detected, the recovery times were in the range of 1.5 to 2.5 seconds for Execution, Daemon, Surrogate Manager and Voter ARMORs. Again, this is a relatively small delay.

As such, Chameleon can be considered quite responsive. As noted, it's flexibility permits differing degrees of responsiveness as required, from indifferent to approaching soft real-time. Additional modifications, such as more stringent resource control, could be added to support hard real time applications.

FAULT COVERAGE

★★★⯪☆

As mentioned in the reliability section, fault coverage is somewhat of an idealistic metric, but is nonetheless important. It refers to the range of faults that can be handled by the system. It is impossible for any system to cater for every possible fault, but it is certainly possible to cover all the important ones.

As has been detailed previously, Chameleon provides coverage for a wide range of faults, ranging from hardware failure to software bugs. Some areas where it is not so strong, however, are:

- Faults in certain key ARMORs, such as the FTM, which can go undetected, or not be detected in a timely fashion. Particularly relating to malicious behaviour.
- Environment & configuration faults. While Chameleon may attempt to use different hosts and architectures, as available, upon failure of a system, it's approach is somewhat haphazard and does not systematically approach the problem of improper configuration, or an incompatible environment.

Nonetheless, overall Chameleon's fault coverage is quite reasonable, given it's level of operation. For finer-grained fault handling, such as errors deep within a given process, the special Chameleon API can be utilised, although ultimately the resolution is limited.

EFFICIENCY

★★★★☆

Finally, one very important metric is efficiency. Chameleon uses multiple hosts, if available, and may employ significant spatial and temporal redundancy in the name of fault detection and masking. The big question is, does it do so efficiently?

Ultimately, this is up to the users of the system. If they choose to make use of the redundancy Chameleon can provide, then obviously efficiency will be impacted. However, users are not required to do so. Indeed, if responsiveness is not a concern, users can use only temporal, as-needed redundancy - in this way, only as many executions are performed as are actually required, given the presence or absence of faults. The Execution ARMOR provides this basic temporal redundancy automatically.

From another angle, how efficient is the inter-ARMOR communication and management? Thanks to the manager hierarchy, with a single FTM (plus backup) and some number of SMs, with only necessary communication between them, the network overhead and processing latencies are minimal. This comes as a trade-off, of course, with other attributes - such as the improved reliability multiple concurrent FTMs would provide.

While the paper does not mention it explicitly, it is conceivable that ARMORs themselves could be run redundantly. This reduces the efficiency of the system, but again only if the user requires it. Unfortunately, the hierarchal nature of the system permeates it thoroughly, which limits how truly fault tolerant it can be in itself.

Thus, in summary, while the system has variable efficiency, it's adaptive nature ensures it is always reasonably efficient for any given requirements. In particular, the approach of Chameleon, with it's strong hierarchy and limited internal redundancy, seems to favour practicality and efficiency over reliability and availability. Whether this is a positive favouring is subjective, based on the priorities and needs of particular users.

Case study: Web service application

As an example of how Chameleon can be used in a real world system, consider a generic online shop-front, for a small business. The components of this system are:

- **Hardware** - Consumer PCs and low-end servers, without dedicated fault tolerance in terms of RAIDs, redundant or hot-swappable hardware, etc. Assume for this example a dedicated T1 line for internet connectivity.
- **Software** - Generic *NAMP system; *nix (Linux, Darwin & MacOS X, in this case), Apache, MySQL, PHP. Additionally, binary-only proprietary middleware for EFTPOS transactions with the business' bank. Consider the Apache & PHP front-end to be run on one machine, the MySQL database on another, and the bank's middleware on a third.

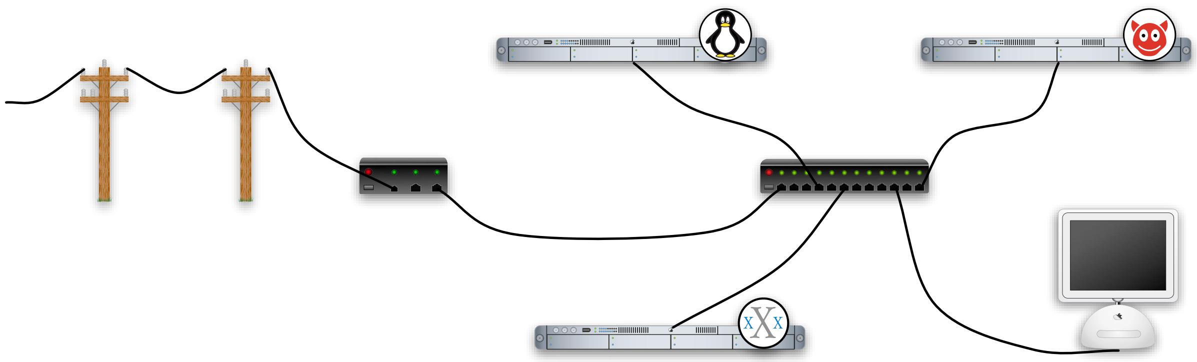


Figure 6 - Example business network

Refer to figure 6. As noted, there are three servers in this environment plus one desktop PC, used to manage the system.

The flow of data in this system is fairly simple - customers connect to the website, hosted by Apache. The online storefront is implemented using PHP, which accesses the MySQL database containing customer information, transaction records, inventory lists and prices, etc. Lots of data is pulled from the database at regular intervals, as customers browse through the store. Occasionally data is written to the database, when a customer modifies their shopping cart, or checks out an order.

The bank middleware is invoked only for each sale. It is invoked via a special plug-in module for PHP, which returns the success or failure of a given transaction. The PHP store code then acts appropriately, e.g. commits the sale to the database, adjusts inventories, and informs the customer of their successful purchase.

There are several different requirements for fault tolerance within this system. Consider each part of the customer process:

1. **Browsing the store**. This involves many copies of the Apache process running concurrently, each serving different customers. Each process runs PHP internally.

Each customer has a database session, but multiple processes for the same customer may be using the database at the same time. All database queries are read-only. The Apache server is primarily I/O bound, waiting on the database and data from the customer.

2. **Adding items to the shopping cart**. This is similar to the above task, except it involves writing to the database.
3. **Customer checks out**. This involves a database read (for the contents of the customer's shopping cart), a call to the bank middleware to perform the EFT, and then several writes back to the database if the EFT is successful.

For the general Apache use, the key is responsiveness and efficiency - there may be many customers, each invoking multiple Apache processes at a time. As such, techniques such as spatial redundancy are not an option.

Assuming the Chameleon implementation in use can intercept all I/O to a process, and can handle a process forking appropriately (as Apache frequently does), the best FTES to use is also the simplest - just an Execution ARMOR for each Apache process. If a process fails, the Execution ARMOR will restart it with the same inputs. Thus, the customer's request will not be lost. There is, however, the possibility of multiple database transactions as a result. For read-only transactions, this is not a significant issue; only one of efficiency. If efficiency is really a big concern, an ARMOR (perhaps a modified Execution ARMOR) could be implemented to cache database transactions for reuse.

Host affinity is also an issue for Apache; customer requests will be destined for a particular network address, so Chameleon is not able to move Apache processes to another host unless it both a) moves *every* Apache process and b) can change the routing so new requests go to the new host. This could be done with a custom ARMOR, although it is no small feat to do properly, without race conditions and other nastiness.

When it comes to the database, reliability is essential, especially for writes to the database. Thus, spatial redundancy with voting is a desirable FTES. Each database process could be run using TMR, for example, with output only to be issued to Apache or written to disk if there is a majority consensus. This is contingent on each MySQL process producing predictable results for a given transaction, given identical inputs, but this shouldn't be a problem. If it is, a modified Voter ARMOR could be created which understands the output and accepts differences between them in certain places (for example, timestamps or randomly generated IDs).

The bank middleware is a special case. It is a proprietary binary blob, which cannot be recompiled or modified, even if a logic (software) fault or error is discovered. In this respect is also probably has forced affinity to whatever particular platform and OS it is compiled for.

Critically, it *must not* issue multiple EFTs for a single transaction; this would overcharge the customer. A failure reported back to the customer is a better solution than risking overcharging. Thus, temporal redundancy is not an option; if the middleware crashes, Chameleon should not restart it *or* terminate the corresponding Apache process; the Apache process should receive appropriate notification of failure and be permitted to respond to the user and use the database as necessary. A note will also need to be made, so that a human can contact the bank and verify whether the EFT occurred or not, and finish the transaction as appropriate. All this may be difficult - the paper does not at any point suggest Chameleon is designed for this kind of application-handled failure recovery. Nonetheless, it could be done, with suitable modifications to the ARMORs involved.

Assume also that the bank middleware encrypts all it's communication to the bank servers. Consequently, for all intents and purposes no two executions of the middleware, regardless of it's input, will produce the same output. Consequently spatial redundancy cannot be employed.

This more or less rules out Chameleon's use for managing the middleware, in general. Chameleon can certainly maintain an overall Execution ARMOR for the middleware, which will relaunch it if *all* middleware processes exit, and may of course monitor the middleware's host, but that is about it.

If the middleware could be modified to support interaction with a Checkpoint ARMOR, then Chameleon would most certainly be useful, as duplicate EFT transactions could be prevented. Unfortunately, as a proprietary binary blob the business cannot make any changes of their own, and the odds of getting the bank to implement such measures, for a single small business, are tiny at best.

This example shows how difficult it is to apply fault tolerance, with a generic system, to as common an application as running a web server. The complex interaction of multiple processes in even a small-scale system makes it very difficult to isolate stages to which fault tolerance techniques can be applied. Certainly, Chameleon can be modified for the specific needs of the system, but then the entire point of it as a generic, off-the-shelf solution is lost.

Future Work

This section is a collection of thoughts on what directions Chameleon needs to explore in future, particularly in relation to it's present deficiencies. Each suggestion is independent, and they are presented in no particular order.

"Sed quis custodiet ipsos custodes?"

"But who will guard the guards?" There is a distinct lack of self-application in Chameleon. While ARMORs that crash can be detected and replaced pretty trivially, and certain interactions with the FTM can reveal problems there, overall there is very little self-management and self-application of fault tolerance techniques.

In particular, the strong hierarchal nature of Chameleon does not lend itself well to fault tolerance. As even the authors of the paper acknowledged, the classic naive approach to fault tolerance is distributed computing. Classic topics in this area include the Byzantine Generals problem, distributed synchronisation - clocks & mutual exclusion, etc - and so on. None of these concepts or problems are addressed in the paper.

For example, the FTM represents a very big critical point of failure. If the backup FTM is not available, not working correctly, or simply not in sync with the primary FTM, chaos can ensue. And when it comes to the decisions made by the FTM - such as how to manage the network, what FTESs to use for new jobs, etc - there is no fault tolerance at all.

It seems it would be a very good idea to evaluate more distributed algorithms for managing the Chameleon network. Given that every host in the Chameleon network has a daemon ARMOR, this would seem a natural place to implement distributed algorithms. It may be that the FTM is not needed at all. SMs still serve a useful purpose as overseers of individual jobs, but also need a look-in as regards their own fault tolerance.

Live-lock

Another approach to detecting live-lock is to provide an API for applications to use to indicate periodically that they are still running. This style of live-lock detection, known as a watchdog timer, is commonly used in microcontrollers and similar devices, and has been found empirically to work quite well. It does not guarantee detection - a process may still be live-locked even if it does regularly check-in with the Watchdog ARMOR - but it increases the probability of detection significantly. Unfortunately, it does require applications to be written specifically for Chameleon.

As noted, reliable live-lock detection seems to be the biggest reliability problem Chameleon currently has. While it's still an open problem to solve, it seems the current implementation stops short of what could be achieved.

Another possible approach could be to trace the processes' execution, if it seems to be running for too long. If it can be proved that the process is in an infinite loop, for example, it can be immediately terminated.

Security

This is, as noted, a major problem with Chameleon - it has no concern for security at all. Here is a system which allows it's users to run arbitrary code on any host in the system. Even if the FTM is protected from abuse, the lack of encryption between ARMORs, and appropriate signing of executable data, leaves the system open for all kinds of attacks.

It seems the system really needs to adopt a solid PKI (Public-Key Infrastructure). All communications between daemons should be encrypted. Daemons should also offer per-host protection; that is, even if a host is part of the Chameleon network, it can be configured to only allow installation of new ARMORs by certain users (or not at all).

Additionally, all Chameleon code should be signed to prevent tampering in transit. This limits the mutability of the code, of course, which is a trade off that must be evaluated in context.

On a different line off criticism, but again security related, much of Chameleon's more advanced functionality - monitoring I/O, for example - implies the need for at least super-user privileges, and probably custom kernel extensions on host systems. These are privileges that are not lightly granted, and will definitely hinder adoption of the system.

A third independent point on security - what resilience is there to malicious behaviour? The paper uses just one example; that if the FTM deletes a bunch of records, when some other ARMOR later tells it that one of those records should really be deleted, the FTM will say it doesn't have that record, and the ARMOR whines to the backup FTM. But this is a condescending example; a malicious FTM wouldn't acknowledge a failure like this, of course. How then does the system detect malicious ARMORs? The Voter ARMOR in particular is a single critical point of failure in this regard. The Byzantine Generals problem indicates that a truly distributed system can guarantee validity of a voting network for up to a certain number of malicious members (less than one third). This would certainly be better than the current implementation, which cannot tolerate a single malicious voter.

Electing the active FTM

A problem with the manager hierarchy as it stands is that while the FTM claims to have top spot, the backup FTM has all it's power *plus* extra - *it can kill any existing FTM*. This posses a rather substantial problem - any failure of the backup FTM can potentially take down a perfectly good primary FTM. It also provides an attractive point of entry for malicious ARMORs.

Again, this comes back to the need for a more distributed solution. Or, a solution which utilises signed code and related PKI to ensure only authorised hosts can operate FTMs.

Heterogeneous Computing

While Chameleon takes special effort to be platform independent, and indeed can benefit from a heterogeneous network by utilising the differences to eliminate environment-specific faults, it also raises questions about redundant execution. For example, the FPUs on the PPC and IA-32 architectures disagree on certain things, such as rounding, precision and so forth. These minor differences can quickly compound over the course of a long computation, to produce substantially different results at the end. Aside from the very real debate as to which one is correct anyway, how can a Voter ARMOR handle this? The paper does not address this issue at all. Another related problem is endianness; similar story.

It seems like Chameleon needs to be more conscious of potential architectural differences, and may need to be capable of running redundant copies of a process on similar architectures.

Single point of failure

If the daemon is the gateway to the network, for local ARMORs, what happens if it fails? Why is it necessary, in a fully networked environment, to only communicate via this daemon? It may have been the original intention to abstract away the networked communication implementation from each individual ARMOR, on the assumption that different users might wish to utilise different network protocols. But in reality there is one dominant protocol - IP, with it's two flavours TCP and UDP - and that is unlikely to change. Thus, there seems little reason to force a single point of failure in each host's daemon.

It would appear to be a superior approach to have each ARMOR communicate with other ARMORs directly, as necessary. The daemon could still remain, as a general monitor of it's host and of course a means of installing new ARMORs on the host, but it's role as the network gateway is unnecessary.

Fanout ARMOR

The principle by which the Fanout ARMOR operates, and for why it exists, is an excellent one - it is critical that redundant copies of a process receive *exactly* the same inputs. However, whether an entirely separate ARMOR is necessary for this remains to be seen. Once proper security is implemented, with data signing and encryption, it is relatively trivial to implement a distributed transaction system like the Fanout ARMOR does, only it could be done instead as part of the Execution ARMOR (or indeed, any ARMOR).

Ultimately, the need for a distinct Fanout ARMOR can be debated, but it seems it's presence in the paper's implementation is largely to workaround larger issues, such as missing encryption and signing.

Atomicity in Surrogate Manager ARMORs

There are some concerns about the nature of certain operations that should be atomic. For example, the SM is responsible for terminating it's ARMORs, once it's task is complete, and then notifying the FTM that those ARMORs have been retired. But consider what happens if the SM crashes after terminating it's managed ARMORs, but *before* informing the FTM. The FTM will still consider those ARMORs alive, as will other SMs. Eventually one will try to use one, and will be unable to. This would be okay, if it were not for the generally suspicious nature of many ARMORs - they may trigger the backup FTMs "fault detection", when it sees that the FTMs picture of the network does not match reality, and thus may erroneously instigate replacement of the primary FTM.

If instead the SM informs the FTM before retiring the ARMORs, but then fails to, the problem is reversed; now there are idle ARMORs sitting around which will never be used.

It seems a simple solution is to have the FTM be the only ARMOR in the system which can retire other ARMORs; thus when a SM finishes, it informs the FTM of the fact, and suggests which ARMORs should be shutdown. The FTM can then, *at it's own discretion*, retire the unneeded ARMORs.

Conclusion

At first glance the Chameleon system looked like typical academia; yet another idealistic system developed without real world considerations. Upon closer inspection, however, it turns out to be quite a reasonable system. That the authors presented it alongside results from a working implementation leads notable credence to it's authenticity and functionality.

And while this document is highly critical of the system in places, overall Chameleon is rather elegant and promising. It's modular design, focusing on code and instance reuse, is simple yet powerful. It is well designed insofar as it only concerns itself with the handful of faults and errors which together lead to the vast majority of failures, and protects again those failures quite effectively. It doesn't try to be everything to everyone; just all it needs to be for most users.

Unfortunately, little more has been heard on Chameleon since the original paper, in 1999; half a dozen other citations in other papers, and little else. The implementation has certainly not taken the distributed computing, fault tolerant, or real time worlds by storm. To be fair, it has some tough competition in all those areas.

On a personal note, I regret not making time to actually implement the system. While some more advanced functionality (e.g. I/O monitoring) is not trivial to implement, the core of the system is actually quite simple. And even just a basic implementation - the FTM, SM, Execution and Voter ARMORs - would be quite enough to gain a large portion of Chameleon's functionality.

Perhaps, sometime down the track, I will get a chance to investigate Chameleon further.

References

[1] Zbigniew T. Kalbarczyk, Ravishankar K. Iyer, Saurabh Bagchi, Keith Whisnant, "Chameleon: A Software Infrastructure for Adaptive Fault Tolerance", *IEEE Transactions on Parallel and Distributed Systems*, volume 10, number 6, pages 560-579, June 1999.

[2] K. P. Birman and R. van Renesse, *Reliable Distributed Computing with the Isis Toolkit*, Los Alamitos, California: IEEE CS Press, 1994.

[3] J. H. Wensley, "SIFT Software Implemented Fault Tolerance", *Proc. Fall Joint Computer Conference, AFIPS*, volume 41, pages 243-253, 1972.

[4] B. Soh, *CSE41RFS Lecture Notes*, La Trobe University, 2006.