

Fault-tolerant systems

Johanna Svenningsson (mea@nada.kth.se)

21st May 2003

Abstract

This report intends to summarize the paper “Understanding fault-tolerant distributed systems” (by Flaviu Cristian) and discuss the papers strengths and weaknesses.

Fault-tolerance is growing more and more important as society and individuals increases their dependence on computer systems. An important goal is therefore to provide computer systems that are fault-tolerant, and ultimately are available to the users, even in the presence of failure, maintenance and growth.

Cristian introduces some basic concepts for analyzing fault-tolerant systems and aims to summarize and bring some order to the current knowledge on fault-tolerant systems. He also makes a suggestion about an automatic service availability manager, which aims to automatically control the existence of the number of servers that are needed for the required level of fault tolerance.

1 What is fault-tolerance?

Fault tolerance means either that a system has a well defined failure behavior or that it has the ability to mask failures to the user. A well defined failure behavior is a failure behavior that follows some well defined pattern. Masking failures means that the system somehow is able to provide the required service even though a failure has occurred.

1.1 Basic concept

When dealing with all kinds of distributed systems one, according to Cristian, needs three basic concepts; servers, services and the depends relation.

A service is the system behavior from user of the systems point of view - the way the system responds to input. A server is the unit that carries out a service. A server u depends on a server r if the server u is a user of a service provided by r . u is then called a user of server r and r is called a resource for u .

One could argue that the concept of servers and services is no longer relevant if the components interact as equal peers. However, both peers expect the other one to provide certain services and thus they become servers to each others.

1.2 Failure classification and semantics

A server is supposed to reply to inputs in a way that is consistent with the server specifications - if it does not there is a failure.

According to Cristian failure can be divided into several different categories.

- Response failure occurs when the server responds incorrectly, either with an incorrect answer or an incorrect state transition.

- A timing failure is when the server responds either to late or to early. If the server replies to late the failure is said to be a performance failure.
- Omission failures occurs when the server fails to respond to an input.
- A crash failure is said to happen if the server after the first omission failure omits to respond to all subsequent inputs. Crash failures can be divided in three categories; halting-crashes when the server never restarts, amnesia-crashes when the server restarts in it's initial state and pause-crashes when the server restarts in the same state as it was in before the crash.

I sometimes let the term omission failure also include halting- and amnesia-crashes, since both of them results in the user not receiving a reply.

Some failure semantics are easier to handle than others. An omission-failure is for example much easier to detect and recover from than a response failure, since you simply need to ask again if you didn't receive a reply while it is much harder to detect if the server is lying or changing it's state in a bad way without telling you.

In analyzing and designing fault-tolerant systems it is of utter importance to know the types of failures that are to be expected. Therefore one has to extend the basic concepts with the systems failure semantics (the failure behavior) when discussing fault-tolerant systems. The fewer kinds of failure a system is expected to show, the stronger the failure semantics for the system is said to be, for example is omission failure a stronger failure semantics than omission/timing.

Choosing the failure semantics is an important part of designing a fault-tolerant system. The stronger the failure semantics, the easier it is to deal with failures in the application. However it can be expensive and hard to make a component have a specific failure semantic. Cristian summarizes this into what he calls one of the fundamental rules of fault-tolerant computing: The stronger a specified failure semantics is, the more expensive and complex it is to build a server that implements it.

Statistics, testing, formal proofs and similar are used to make sure that the chances of a failure occurring that is not a part of the failure semantics is negligible.

1.3 The concept of masking

If a system masks a failure it hides the failure from the user. There are two ways of doing this according to Cristian, hierarchical masking and group masking.

In one way or another most computer systems are hierarchal, that is they are organized in several different abstraction layers and one layer is always depending on resources in lower abstraction layers to provide its services. For example is a webserver dependent on resources such as CPU and memory.

If one of these resources fails, the server will also fail if there is no mechanisms to provide fault tolerance. However, it is not certain that the server will have the same failure semantics as the resource, even if a user of a resource mostly will show an arbitrary failure semantics if the resource does so.

In order to provide fault-tolerance the server needs to detect and handle failures of the resources and either propagate a "nice" failure to its users or somehow provide the requested service in spite of the failure. This is in hierarchical masking done by the server detecting the error and either using another resource or returning an exception.

Another way of implementing fault-tolerant service is by assigning a group of servers to provide required service. Depending on what failure semantics the members of the group is expected to have, the group output can be generated in several different ways.

If the members have an omission failure semantic the group simply needs to make sure one member of the group replies to each request, for example by ranking the members according to speed and letting the fastest member reply. In this case a group with $2n$ members will be able to mask failures from $2n - 1$ members of the group.

On the other hand, if the members have arbitrary failure semantics some kind of voting protocol is needed to make sure that the output is correct. Voting algorithms mostly need a majority of the members to vote, so in this case failure can only be masked for $n - 1$ members of the group.

In general, the weaker the failure semantics of the system is the more complex and expensive group management mechanisms are needed. This is called the other general rule of fault-tolerant computing by Cristian.

2 Fault-tolerant hardware

The goal for fault-tolerant hardware is to build hardware so that units can fail, be removed or placed into the system without disrupting the users of the system. This might according to Cristian be very expensive or impossible to achieve, so a more easily reached goal is to give the hardware a well defined failure semantics.

There are two very common strategies for ensuring crash or omission failure semantics in hardware. Error detecting codes can be used in order to detect failures that corrupt data, for instance in busses, memory and disks. Duplicating and matching can be used in all forms of hardware to detect failures.

In order to mask hardware failure it is possible to use redundant units and handle the failures at a hardware level. It is also possible to let the operating system detect the failures and make sure that other resources are used.

3 Fault-tolerant software

The ultimate goal for fault-tolerant software is being able to remove servers without disrupting the activity of the users. If this is too expensive or hard to achieve the second goal is to make it have a “nice” failure semantics in order to make users able to recover from the failure.

When implementing fault-tolerant systems the software is commonly assumed to be totally or at least partially correct. The definition given by Cristian of a totally correct program is that it behaves as specified as long as the services it uses do not fail. A partially correct program may suffer crash or performance failures, but it does not lead to any incorrect state transitions. Partial correctness is interesting because it is easier to achieve and easier to test. Since a partially correct program keeps its partial correctness even in the presence of failure of its resources it leads, according to Cristian, to a quality assurance goal for all abstraction levels:

Make the hardware have crash or omission failure semantics and make the software be totally or at least partially correct.

3.1 Synchronization and replication policies

Cristian defines three kinds of policies that according to him are useful when designing fault-tolerant systems; the synchronization, replication and availability policies.

The synchronization policy prescribes how far apart the servers' states can get. The replication policy for a service s specifies how many servers should exist. For example a replication policy of 2 specifies that 2 redundant servers should be

used to implement s . The synchronization and replication policies specified for a service form the availability policy for that service.

There are, according to Cristian, two main categories for synchronization in a group of servers. Close synchronization requires all servers to execute the same series of state transitions in the same order. Loose synchronization does not require all the server to have the correct current state - they should, however, be able to reconstruct it.

There are several different ways to get output from a closely synchronized group. One way of doing it is simply letting all the members reply as fast as possible and letting the output be the output from the fastest member. This will however create a large communications overhead. Another way is to rank the servers and let the highest ranked member be the one that provides the output. A group that behaves this way will be functioning as long as there is a working fastest member and there is a nice failure semantics in the group. If, however, the group has an arbitrary failure semantics some kind of voting is needed and the group would be unable to provide its service if a majority fails.

Loose synchronization ranks the members of the group by how closely they are synchronized to the current service state. The highest ranked one is usually the one that handles communication with the user and updates the state. The other servers act as backup servers that only receive state updates and record the requests. The primary advantage with this model is that it is possible to use the backup servers for other purposes as well, since they do not make as much use of their resources. The main drawback is that the delays seen by the clients are longer than seen with close synchronization.

A replication policy prescribes how many members a group is expected to have. The more members - the greater the availability. The number of members is also connected to the communications overhead, so the number of servers in the actual systems has to be balanced between high availability and low communication costs.

Cristian also defines something he calls an availability policy, which is constituted by the replication and synchronization policies. In Cristian's syntax $\{loose,2\}$ would mean a server/backup pair (also sometimes called primary/secondary).

3.2 Automatic enforcement of the availability policy

In order to create fault-tolerant systems it is a quite natural goal to make the system handle as much of the configuration and reconfiguration as possible on its own.

Cristian introduces two different ways of doing this. The first way he mentions is to include all necessary mechanisms for detecting failures, promoting someone to act as primary and handling new groupmembers in each member.

The second way he talks about is by extracting the tasks concerning failure detection, coordinating promotions and enforcing the replication policy into a separate service availability manager service.

Cristian sees the service availability manager service as a group of servers that are dedicated to enforcing the availability policies for several different services. The service availability service is able to keep track of possible hosts for a service and start/stop servers on them. In the case of loose synchronization it is also responsible for promoting secondaries.

According to Cristian, the second solution is preferable to the first since it will reduce code duplication and increase modularity, since the second solution does not require all services to implement mechanisms that are basically the same.

3.3 Discussion

The paper is intended as an overview of the field of fault tolerant computing and as such it does not go into details very much. In some ways this is good since the paper is quite long already, but it would have been better if Cristian had elaborated some parts a bit more.

One such field is the reasoning behind the intended purpose of the availability policy and the way he chooses to define it. In the case of closely synchronized systems the availability policy does not say anything about how many members that can fail, only how many servers are being used. For example could {close,5} both mean a group where the fastest member replies or a group in which voting is in use. The number of members that can fail can thus be either 2 or 4 with the same availability policy.

Another thing that is unclear about the availability policy is connected to systems with a distributed global state. In order to achieve fault tolerance it is not necessary to distribute the whole state to all servers; different servers can be responsible for different parts of the global state and these servers can both be loosely and closely synchronized. In such a case an availability policy described like {loose,5} would be quite uninteresting with Cristian's definitions, since it would not say anything about how many times the state is replicated and thus it does not say anything about how much redundancy that exists in the system.

I consider this to be interesting, since load balancing is an important part of today's distributed systems. It might be interesting to let the availability policy somehow contain information about how many times the state is replicated or how many servers that can fail without the service becoming unavailable. However, this would be more complex and difficult to understand than the very simple definition proposed by Cristian. It might also be unnecessary, depending on the intended purpose of the availability manager.

The idea behind an automatic availability manager is a nice thought. The availability manager suggested by Cristian, however, only enforces the replication policy in a complete way. In the paper Cristian indicates that it would be relatively easy to let the availability manager enforce the synchronization policy as well, but I do not believe that this is the case. The synchronization largely has to be done from within the servers - they are the ones with the data that needs to be synchronized and with specific requirements about how this must be done. Specifying the synchronization policy from the availability manager will, in practice, not be much different from giving other kinds of arguments to the programs if not a very extensive knowledge about the applications and their way of synchronizing is built into the availability manager. This would make a general service availability manager very complex and the more complex a program is - the more prone to errors it will be.

Thus it might be more interesting to only make a general replication policy manager, since the way the replication policy is to be enforced is quite similar for most (if not all) fault-tolerant software.

4 Summary

As society increases its dependence on computer systems the need for systems that are able to provide their services even in the presence of failures increase. Thus it is important to increase the understanding and knowledge of fault-tolerant systems and make fault tolerance easier to achieve.

When talking about fault-tolerant systems one needs to use both the concepts that are used in all distributed systems and those that are specific to fault-tolerant systems.

One of the most important among the concepts related to fault tolerance is failure semantics - the way the system fails. When designing a fault-tolerant system one wishes to have as strong a failure semantic as possible, however it might be difficult and expensive to achieve this. Another of the important concepts is masking failures, that is, being able to provide the requested service even in the presence of failure.

Cristian also defines three policies that are used to describe fault tolerant systems. The replication policy prescribes how many servers that are to be used. The synchronization policy describes how these servers should be synchronized. Together they make up the availability policy for the system.

I think Cristian succeeds rather well in giving a set of basic concepts for analyzing and designing fault-tolerant systems. Some parts, however, still needs to be clarified, for example the intended purpose of the availability policy.