White Paper on High Availability Design

Introduction

Communication systems and applications are getting deployed very widely in the public and enterprise networks. Systems that are used by a large user population on a public network are normally expected to provide a high degree of service availability. In the absence of reliable service, end user experience gets frustrating. Compare the user experience of the very reliable telecom network with the relatively less reliable Internet systems. End user expectations are slowly rising. If serious levels of customer service is to be provided, the availability levels of service has to be very high. In this white paper the issues involved in improving the availability of service are discussed with special emphasis on software based techniques.

Carrier Class Availability

Network elements, such as telephone switching systems, typically operate with a target availability of 99.999 % - often referred to as "five nines availability." This level of availability translates to the equivalent of a telephone switch being allowed to be out of service for only a few minutes per year. These few minutes per year include all of the time needed to repair faults, load software, upgrade hardware, and perform periodic maintenance and any other necessary activities. High availability assumes that active calls are not lost during switching system failure and that only a small fraction of calls in progress may get handled incorrectly. However currently, Internet, other broadband services, and many other systems do not all enjoy this level of availability. Telephone switching provides an availability benchmark against which to measure other products and services. A complex switching systems contains 100s of PCBs, 1000s of CPUs, Millions of lines of code providing many complex features and handle large loads. Still Central office designers have managed to achieve and demonstrate very high availability. There are many lessons that can be learnt. Today Wireless and IN system designers are coping with the same issues central office designers dealt with a few years back.

It has been recognized that a large part of the system unavailability has been attributable to software defects and procedural errors. The hardware reliability has increased significantly over the time by improved use of technology and by use of active and standby copies and distributed system architecture using various schemes of redundancies. However the software is typically not redundant and requires a systematic effort to increase the availability.

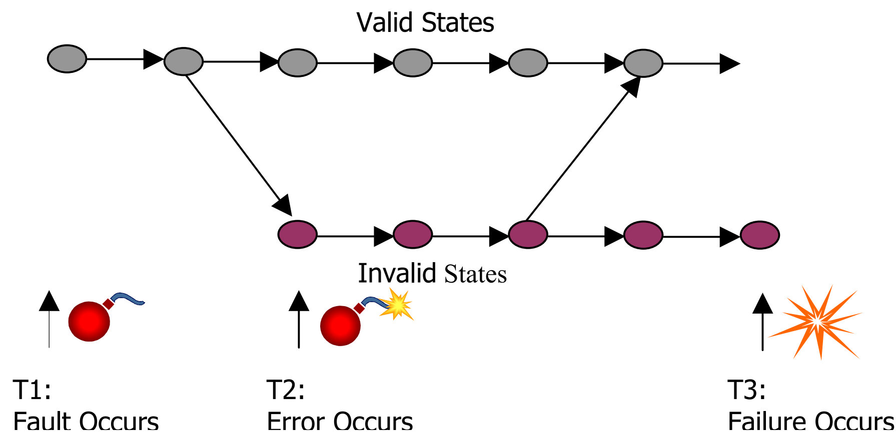
Some key concepts and terminologies

Reliability and Availability are complementary concepts. The reliability of a system is the conditional probability that the system will operate properly during a specified period of time. The availability of a system is the probability that the element is in service and available to a user at any instant in time.

Systems may go out of service for any number of reasons, such as the occurrence of a fault, repair activities, software loading, hardware upgrading, or periodic maintenance. For a system to achieve high availability, the duration of these interruptions must be as short as possible. It will be highly desirable to handle these situations without any loss of service to the end user. A system may be considered highly reliable (that is, it may fail very infrequently), but, if it is out of service for a significant period of time during a failure, it will not be considered highly available.

Fault, Error and Failure are related and occur chronologically one after the other. Fault is an erroneous state of software or hardware caused by an incorrect design, component failure or user error. Error is a manifestation of the fault. An error puts the system in a state from which a series of valid transitions can lead to a failure. An Error occurs later than the occurrence of the fault. Failure occurs when the behavior of the system first deviates from the specifications. Failure follows an Error.

The diagram below brings out the chronological relationship between Fault, Error and Failure in any system. A system keeps making transition from one state to another while providing useful service.



Fault Occurs when a defect is created. The fault may be due to a component failure in hardware or a defective program design. When the defective programming was done, the fault can be said to have been injected in to the system. When the faulty code gets executed, it may cause the system to enter into an erroneous state. For example, it may result in assigning wrong data to a variable. Now the system is said to have entered an erroneous state. Depending on when the erroneous data gets used, the system may exhibit a "failure" in terms of violating the behaviour as expected in the external specification of the system. The failure may occur immediately after the erroneous state was entered or hours or days afterwards. Fortunately, there is a time latency between Fault, Error and Failure. A high availability system must take advantage of this fact in building fault tolerance in the system. An erroneous state is detected before it leads to a failure, and the system restored to a valid state.

Failure management stages

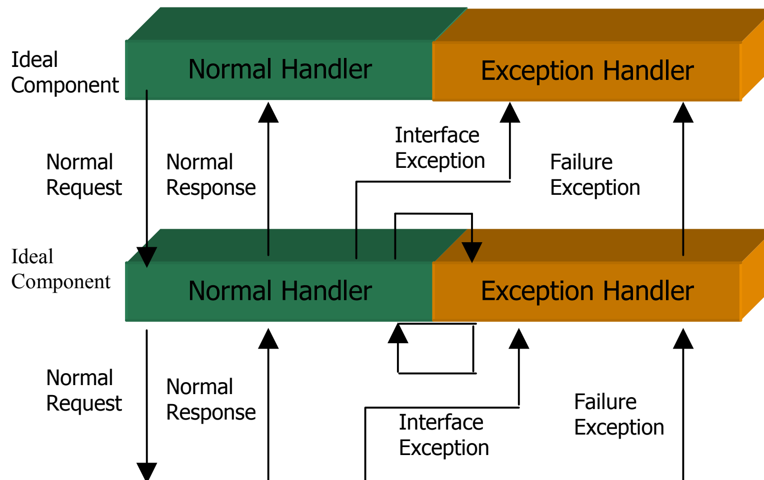
High-availability systems minimise the effect of faults by using well-defined stages of fault management. The stages followed typically are: fault detection, localisation, confinement, recovery or masking, cleanup, repair and restoration. Fault detection is done by combination of techniques that include Replication checks, Timing Checks, Reversal checks, Coding checks, Reasonableness checks, Structural checks, Diagnostic checks and Defensive checks. Failure localisation uses some of these checks again to resolve the location of the fault so that a right strategy can be adapted to recover or mask the fault. Confinement ensures that the effect of the

defect does not spread across the system. Recovery or masking is essentially bringing the system back to a valid state. This may be achieved by completely recovering from the error or hiding the ill effects of the error. From an external user's point of view, the system has managed to mask the error. It is also possible to use a strategy to reduce the severity of the possible failure by using graceful recovery strategies. Cleanup takes care of ensuring that no unintended side effect of the problem is still lingering in the system. Repair and restoration in the case of hardware defects will involve a replacement of the faulty component or subsystem to restore the system to its original status. In software, this step may involve analysing the data by the designers and eventually fixing of the problem. This step is not part of the immediate action on managing a fault. Systems are known to provide high availability by masking the errors in the system for long periods without the software defects getting fixed.

Techniques and Strategies

A number of techniques and strategies are used in ensuring the high availability of the system as per the fault management stages described above. A few of them are described here.

In achieving increased tolerance to faults in software the framework of an **Ideal Component** is a powerful abstraction.



The system can be visualised to be built in terms of ideal components. All subsystems, layers, partitions, modules or components that make the system can be considered to be ideal components. While a really ideal component as the name suggests will be an elusive goal, the

concept is very powerful, because it articulates a framework in which every component will make all reasonable attempts to be robust. An ideal component is fault tolerant and makes all reasonable attempts to recover from exceptions encountered within or from lower level components. It also will not allow invalid data from its clients to introduce errors in its own state. An ideal component is thus a robust element of the system. When a system is built using ideal components, robustness of the system as a whole improves significantly increasing service availability.

Error counters and threshold checks ensure a continued measure of the health of the system. Excessive error counts will act as error detection mechanisms to initiate fault management steps. **Audits** of data structures have been found to be very useful in effectively masking software errors. **Time stamped resources** is one of the techniques used to verify the health of resource pools. It is highly desirable to ensure that the design consciously avoids any flurry of activity take place. A continuous stream of events can completely swamp the system. This is best avoided by looking for every such source in the design and introducing some flow control. **Watch dogs –** both in hardware and software is a very useful strategy to detect and initiate recovery from many deadlock conditions. **Time outs** are useful if used in the context of the ideal component behaviour. Systematic **Interface checks** across module and subsystem boundaries are very useful. It is generally possible to protect the ill effects of one module or subsystem from spreading to the other. In the absence such measures the error spreads like cancer within an unprotected system making error masking virtually impossible. **Graceful recovery** strategies help ensure minimal impact on external users. The recovery actions are structured in increasing levels of severity so that the all possibilities of restoring with minimal damage are fully explored before taking the most severe recovery action like restarting the system. **Avoiding Human Intervention** is a cardinal principle of high availability. The recovery strategies of the system must never assume some steps to be taken by the operator or user. The human role must be confined to repair only. Restoring the service has to be fully the system's own responsibility. A **task restart** capability is an useful strategy to achieve localised recovery even within a module. Typically this strategy asks for all tasks to be able to unwind themselves from wherever they are, reinitialise themselves fully and be ready to provide continued service to others. **Check pointing** is an very powerful technique wherein sufficient information is remembered by the system to fall back to when encountering a problem.

Keep them Simple. These and other techniques are powerful and very useful in practice, if they are kept simple. More complex the strategy is, the less likely the chance of its success is.

Conclusion

Increased service availability is increasingly becoming a software problem. Simply providing fault tolerant hardware architecture is not enough. Simple yet powerful techniques, strategies and frameworks have worked very well in high availability telecom systems especially in the Central office switching. It is possible to extend these proven strategies to the newly emerging systems, which offer service to a large public and build systems that "run forever".

*The author **Mr. R.Venkat Rajendran** was the head of the Fault tolerance and high availability design group for a large Central office switch and eventually headed the entire software design for switch for over a decade. He has innovated a number of techniques and frameworks, which have been successfully demonstrated by more than 20 Million lines of fixed line systems, deployed providing high availability services. Mr. Rajendran now heads the software design activities at **Deccanet Designs Ltd**, a company co-founded by him. He offers Telecom software design services with an increased focus on building reliable and highly available software systems for Telecom.*