Building High Performance & Highly Available Systems

by Jeff Garland
President & CTO
CrystalClear Software, Inc
August 2000

Overview

Over the past 15 years I have been involved in the development of several performance critical and highly available systems for telecommunications, process control, securities trading, and satellite management. All these systems have the goal of reaching at least 99.999% availability. In addition, they all have elements that require high performance. Some of these systems have pieces with hard real-time constraints while others simply have a need for high transaction throughput. Several of these systems are widely deployed and hence they provide examples of successful highly available and high performance systems.

This paper will describe some common features and mechanisms of these systems that result from high performance and highly available requirements. Second, it will describe how these features impact the software architecture. Finally, it will attempt to distill some key approaches for building robust architectures for high performance and highly available systems.

To understand how performance and availability impact software architecture, we must first define architecture. The 4+1 view model of architecture by Kruchten [Kruchten98] defines 5 main aspects of software architecture: the logical view, the implementation view, the process view, the deployment view, and the use case view. The logical view addresses the system functional requirements and provides both static and dynamic views of the classes and packages that comprise the system. The organization of the source code, data files, and other aspects of the development environment are modeled in the implementation view. The process view models concurrency issues, such as splitting of functions into processes and threads. The deployment view describes how the various components are mapped to hardware at run time. Finally, the use case view describes a set of scenarios that the system must support. The following discussion will focus mostly on impacts to the logical, process, and deployment aspects of the architecture.

Common Features and Techniques of High Performance & Highly Available Systems

Table 1 summarizes some common features and techniques utilized in high performance and high availability systems. These features and their architectural impact will be described in detail below. Each feature is a result of the performance goals, availability goals, or both. In some cases, the performance and availability goals conflict. That is,

the utilization of a particular technique for performance may confound availability or vice versa.

Feature/Technique	Driving Factors	Conflicting Factor
Self-knowledgeable (configurable, instrumented, and upgradeable)	Availability, Performance	
In-memory caches & databases	Performance	Availability
Isolation of real-time processing	Performance	
Loosely coupled distributed processing using message oriented communication	Availability	Performance
Multi-layer error handling	Availability	

Table 1: Common Features of HA & HP Systems

High performance and high availability often requires software that is self-knowledgeable. That is, there is a suite of management software that is aware-of and operates on the software itself. This knowledge usually includes at least three dimensions: configuration, instrumentation, and online upgrade. Configuration is frequently used to for performance and scalability, online upgrade is required for availability, and instrumentation is required for both performance and availability.

A configurable system allows for "coding free" optimization of system functions. For example, different installations of a software system might need to scale from a couple of nodes to fifty or more. The software components are not typically re-written, but only reconfigured to take advantage of the additional processing power. Elements of the system utilize the configuration to perform system startup, monitoring, load balancing, and other administrative system tasks. Many systems support the ability to change the configuration and hence the processing patterns without restarting the system. Building configurable components impacts the logical architecture (requiring components to be have configurable aspects) and often reflects the physical aspects of the architecture (nodes and network in the system). Recently, these techniques have become widespread as the rise of the Internet and various directory services. For example, the Domain Name Service has demonstrated the flexibility and scalability that can be created by building software that allows for reconfiguration of the physical architecture.

Instrumentation supports monitoring of status, performance tuning, and often online debugging. Status instrumentation allows the system to monitor components for failure. Human operators may be provided with alarms that alert them to perform maintenance tasks when a failure occurs. Performance instrumentation provides statistical data that allows for overall system tuning. Online debugging can provide software tracing and other features to assist in problem resolution. Instrumentation tends to crosscut the logical architecture of the system, affecting the design and implementation of many components.

Online software upgrades are often essential to providing high availability. Upgrading software without shutting down the system can be extremely complex. It often involves partial system shutdowns, online database evolution, and simultaneous operation of multiple versions of the software. It is often a requirement to be able to rollback to the old software version if something goes wrong with an upgrade. The design of new versions of software may be significantly limited since outage time might be required for some complex upgrades. For example, database schema upgrades might be put off until a major release. Temporary patches, which utilize existing database fields in non-standard ways, may be required to work around the inability to upgrade the database schema. Online upgrade adds a whole new dimension to software design and management that is not well supported by the current breed of software production tools and methods.

In-memory databases and caches are a common technique in high performance systems that must provide bounded time or extremely fast response. The in-memory database overcomes I/O speeds that are too slow to meet performance goals. Clearly, the in-memory database is at odds with the highly availability goal since the memory is a "volatile" single point of failure. For high read, low write databases (a simple website database or system configuration) the cache is easily distributed and writes can be logged synchronously to disk. In systems requiring high write volumes, the in-memory database is usually made highly available by logging to disk, redundant hardware, or a distributed backup. Traditionally, in-memory databases have been highly customized solutions with a significant impact on the system design and architecture. Recently commercially available in-memory systems (e.g. Times Ten) have begun to offer a more standard alternative.

Often in hard real-time systems with bounded time constraints on processing, the real-time aspect of the system will be logically or physically isolated from the non-real time aspects of the system. For example, a real time operating system may ensure that certain functions are performed at a regular interval (ie: polling hardware to detect state changes). Once state changes are detected this fact is then sent as a message to some other processor for display or other processing. Thus, low-level event detection is isolated from high-level processing.

The most significant influence of availability and performance requirements is in the deployment aspect of the architecture. A key principle of building high availability systems is to avoid single points of failure. As a result, all processing and state must be distributed among multiple physical hardware nodes or onto highly available hardware. Today, more and more systems are attempting to forgo the cost of high availability hardware by implementing fault tolerance in software alone. As an end result, distributed processing becomes essential to the implementation of availability goals. Even systems that utilize high availability hardware to simplify software design often use distribution to provide the required performance scalability. Many systems utilize a "loosely coupled" distributed approach. That is, a set of software components connected by a messaging infrastructure. A reliable publish/subscribe paradigm is a common strategy for communication between components. Of course, distributed systems introduce a host of

other issues including the need for reliable networks and replication or logging of state information.

Successful high availability systems tend to have a multi-level error handling approach. For example, code must be able to detect and efficiently handle invalid data without application failures. In addition, when an application detects an unrecoverable condition (usually some sort of software bug) it must be capable of rapidly restarting to an uncorrupted state with a minimal impact on the system. When an application crashes it must provide debugging information so that software problems can be rapidly addressed. Design of robust error handling schemes is a difficult task that can significantly impacts the logical architecture and component interfaces.

Strategies for Building HPHA Systems

The software architect should strive to minimize the impact of performance and availability requirements on the logical architecture. Most software is difficult enough without worrying about how to make it fault tolerant and high performance. Ideally, a small focused group dedicated can address the high performance and high availability aspects of the system. For example, many application displays, which are not impacted by these requirements, can be developed more effectively without a focus on performance and availability. Overall, application development is clearly simplified when it does not need to be highly concerned about performing some set of processing within a set deadline (hard real-time). An object-oriented analysis model developed without regard for the process and deployment architecture often provides a good starting point for building a robust architecture. This allows the team to get an understanding of the logical architecture without the additional complications of process and deployment. Sketches of the expected deployment architecture should also be created during the development of system analysis and requirements to begin understanding the constraints and limits that will be eventually be imposed on the design. Finally, as the design progresses process and deployment design need to be a key focus to ensure correct mapping of functions to components.

In the area of performance, it is critical to get an early understanding of the performance requirements. In addition, design of hard real-time elements should be isolated. An early understanding of the performance requirements allows for early prototyping to mitigate potential design issues. Of course, often the details of particular aspects of performance may not be well formulated during initial design. For example, typically in communications systems there are well-understood limits to the allowed delay time before a human ear can perceive a delay in a voice conversation. During the initial design it is known that various components must perform a function that is bounded by this delay time. However, it may not be clear how to assign budgets to these components. Further, even if performance budgets are assigned it may not be possible to implement within these budgets.

Almost all high performance and highly available systems are distributed systems. A loosely coupled approach that utilizes a messaging infrastructure seems to be a recurring solution. This solution provides for separation of components with well-defined interfaces. In addition, it can build highly scalable systems since new processing components can be easily added. In addition, commercially available messaging products and standards eliminate the need for system designers to recreate this fundamental aspect of the system. It is also desirable if processing units of the distributed system can be stateless (ala a simple web server). That is, there is no processing state to backup for fault tolerant operation. This strategy is ideal since it allows for both load balancing and fault tolerant operation. However, many applications must provide replication of state by taking snapshots at various processing points. However, building a design that minimizes the need to replicate state simplifies software upgrades and fault tolerance design.

From a software development perspective it is critical to have highly evolved software production processes. The ability to rapidly provide and document software changes is a key to success in achieving fault tolerance. Design and code inspections are usually needed to reduce the number of bugs and ensure the implementation conforms to the architecture. Highly iterative processes are usually essential to meeting high performance goals and stabilizing the architecture as early as possible.

Using off the shelf components can be both helpful and harmful. Some components are made explicitly for these kinds of applications and provide a big boost to the speed of development. However, some components may not be well suited for high performance or high availability systems. It is crucial to carefully evaluate components to be sure they will not fit into the system configuration, error handling, and instrumentation schemes. If a third party product is to be used in a performance critical system component it is essential to understand its performance weaknesses early in the design.

In conclusion, a sound software architecture is essential to the delivery of a high performance and highly available system. This paper has illustrated a few features and techniques of some successful systems and recommended some practices for the development of new systems.

References

Kruchten98 - Kruchten, Philippe, *The Rational Unified Process: An Introduction*. Addison-Wesley, 1998.

About the Author

Jeff Garland is a Software Architect and CTO of CrystalClear Software. He has worked on high availability and high performance systems for Motorola, Honeywell Industrial Automation, and AG Communication Systems (part of Lucent). He is currently re-architecting the Service Delivery Platform for the NASDAQ stock market. He holds a

Master's degree in Computer Science from Arizona State University and a Bachelor of Science in Systems Engineering from the University of Arizona.