2

Overview

This chapter provides a high-level view of extreme software systems. It begins by defining some basic terminology. It then provides a reference model to describe the high-level architecture of a typical extreme system. Next, it defines the concept of a programming model and how such a model relates to the programming techniques that are the topic of this book. It then presents a pattern language that summarizes all of these carrier-grade techniques, and it finally concludes with an overview of the C++ classes that appear in the book.

2.1 BASIC TERMINOLOGY

This section defines some commonly used terms as they relate to this book.

System: hardware and software that is sold as a complete package and that provides an integrated set of capabilities that users value.
Network: a set of systems that interwork to offer users a broader range of capabilities or to serve a larger set of users than a single system can support.
Customer or **operator**: a firm that operates a network.
Subscriber or **user**: an end user of a network.
Service or **application**: a capability that subscribers use.
Craftspeople: customer employees who monitor and operate a network.
Engineering: configuring a system's hardware and software resources so that it provides acceptable service to subscribers when running at maximum capacity.

Robust Communications Software G. Utas
 ISBN: 0-470-85434-0 (HB)

Designer or **developer**: someone who writes software for an extreme system.
Node: a processing element in a system.
Shelf: a rack that houses a system's hardware.
Card: a circuit pack inserted in a shelf.
Fault: the root cause of a hardware or software error.
Error: improper behavior that arises from a fault.
Failure: an error that causes a hardware or software component to become unavailable.
Bug: a software fault that causes an error or failure.
Outage: a failure that prevents some or all subscribers from using services.

2.2 EXTREME SYSTEM REFERENCE MODEL

The high-level architecture of many extreme systems consists of the following types of nodes (see Figure 2.1):

- An **administration node** provides interfaces for craftspeople. They populate it with configuration data and subscriber profiles, which it downloads to other nodes. It receives status information from other nodes and makes it available to craftspeople.
- A **service node** runs applications on behalf of subscribers.

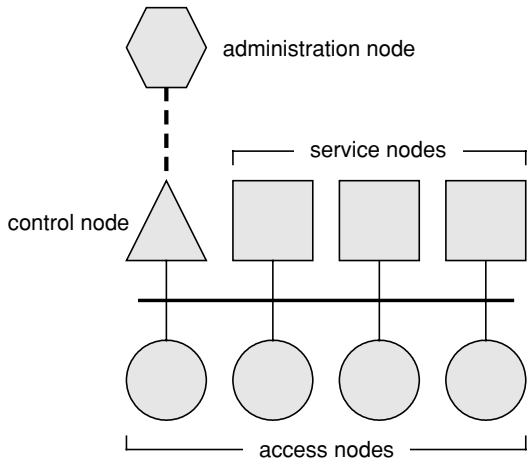

Figure 2.1 Extreme system reference model. The administration node usually resides on a separate platform and communicates with the rest of the system through the control node.

- An **access node** hosts subscriber interfaces, links to other networks, and other hardware devices.
- A **control node** runs system maintenance software that monitors the health of service and access nodes. When a node fails, the control node initiates corrective action.

The system contains one control node and a number of service and access nodes. It is managed through an administration node which sometimes acts as a front-end to more than one system. An administration node is sometimes referred to as an *element management system* if it front-ends one system, or a *network management system* if it front-ends many systems.

The control node often acts as an intermediary between other nodes. It relays administrative commands from the administration node to service and access nodes. In the other direction, it relays status information from service and access nodes to the administration node.

The administration and control nodes provide central points of control for the system. The administration node provides the external world with an integrated view of the system, and the control node contains the integrated internal view. Placing the administrative and control functions in separate nodes removes background work from service and access nodes, allowing them to focus on running services for subscribers.

Service and access nodes are often diskless. Disk accesses by their applications would unduly degrade capacity, and omitting a disk reduces costs. The data and software loads required by service and access nodes then reside on the control node's disk, and service and access nodes download it.

Although the administration node is an important part of an extreme system, it is not itself extreme. Its availability, reliability, scalability, and capacity requirements are not as strict as those of other nodes. Consequently, administration nodes usually run on general-purpose computing platforms, such as commercially available servers.

Service, access, and control nodes face extreme requirements and therefore run on hardened (high availability) computing platforms. Service and control nodes only require pure processing platforms, but hardened versions of these are available commercially. However, access nodes that support specialized hardware interfaces often run on proprietary platforms. The platforms for control, service, and access nodes are typically cards rather than standalone boxes. The cards reside in a shelf or, in a large system, in a set of interworked shelves.

2.3 PROGRAMMING MODELS

Taken as a whole, the techniques covered in this book comprise a programming model for producing carrier-grade software. A **programming model** is a set of proven techniques that all of a system's software components follow. A programming model fosters desired attributes, such as quality or simplicity, and makes it easier for components to fit together. It also provides a consistent terminology, which makes it easier for developers to communicate about their designs.

When a component deviates from the programming model, there must be a good reason for it doing so. Most often, the reason is that there was a trade-off between, say, capacity and productivity. And in the setting in question, the designer made a decision to favor one over the other.

By way of example, a programming model might contain the following elements:

- heap-based memory allocation with garbage collection (Java, for example);
- synchronous remote procedure calls (CORBA, for example);
- preemptive scheduling, using semaphores to protect critical regions;
- priority scheduling;
- spawning short-lived tasks (processes or threads) to perform work;
- frequently reading from, and writing to, disk;
- using virtual memory.

This programming model is a common one within much of the computing industry. It is also *dead wrong* for extreme systems.

What heresy!

Indeed. Yet each of the above techniques compromises carrier-grade attributes in some way. An extreme system based on the above programming model will suffer. It will exhibit less availability, reliability, capacity, scalability, and productivity than if it were based on other techniques that we will discuss.

Can an extreme system be built using the above programming model? Perhaps, but only because, as the saying goes, 'Anything can be done in software'. Some extreme systems actually do employ some of the above elements, but one can liken them to a dog walking on its hind legs. It *can* be done, but it isn't easy or elegant, and it's truly a miracle that it can be done at all.

Administration nodes, however, may freely use things that an extreme system avoids. They may do so because, as mentioned in the

previous section, they do not face extreme requirements. They primarily manage databases and provide graphical user interfaces. Things like Java, CORBA, and virtual memory are often effective in such settings.

If you work, or have worked, on extreme software, you will probably be familiar with some of the techniques in this book. However, if you are primarily familiar with a programming model similar to the one outlined above, some of the techniques in this book may strike you as bizarre. Consequently, we will always discuss the rationale that underlies each of these techniques. This should help you to make trade-offs when you believe that other approaches would be more appropriate.

All the techniques in this book have been proven in carrier-grade products designed by firms such as Lucent, Nortel, and Ericsson. In many cases, they have arisen independently rather than as the result of someone bringing in techniques learned in a previous job. They have often arisen through a process of evolution, during the search for solutions to chronic software problems. However, they have never been documented in a cohesive manner. Instead, they are lore among those who develop products that depend on them. This book attempts to elucidate what is currently a black art.

2.4 A PATTERN LANGUAGE FOR CARRIER-GRADE SOFTWARE

Although this book does not use a formal pattern format to describe extreme software techniques, this section introduces these techniques as if they were patterns, using the format of a pattern language. The figure on the inside cover of this book summarizes this pattern language.

Using object orientation to build extreme systems is desirable because it allows their protocols and state machines to make use of polymorphism and inheritance. However, indiscriminate use of object orientation leads to performance problems. Well-known patterns such as SINGLETON and FLYWEIGHT mitigate this outcome, as do CACHED RESULT and EMBEDDED OBJECT. Singletons often reside in a REGISTRY that implements a POLYMORPHIC FACTORY, which preserves partitioning by allowing high-level software to delegate the instantiation of leaf classes. Defining a top-level `Object` class provides all objects with basic functions, such as `Display`, which are useful in a variety of situations.

Allocating objects from a heap at run-time leads to memory fragmentation. OBJECT POOL avoids this outcome by preallocating object

blocks when a node initializes. Because this provides greater control of object management, it enables further techniques that improve performance. OBJECT TEMPLATE speeds up instantiation by initializing each object with a block-copy operation. When an object usually behaves as a singleton, its object pool uses QUASI-SINGLETON to allocate blocks efficiently. OBJECT MORPHING dynamically changes an object's class to avoid the overhead of reconstructing it. OBJECT NULLIFICATION detects references to stale objects by invalidating deleted objects.

Preemptive scheduling is undesirable for many reasons, the foremost one being that it forces the extensive use of semaphores to protect critical regions, something that is highly error prone. COOPERATIVE SCHEDULING eliminates this risk by providing a global lock that allows each transaction to run to completion. RUN-TO-COMPLETION TIMEOUT guards against transactions that run too long. A `Thread` class provides a WRAPPER FACADE and THREAD-SPECIFIC STORAGE for native threads, along with functions that support cooperative scheduling.

Under cooperative scheduling, applications avoid blocking operations. When this is impossible, a THREAD POOL is used to provide greater throughput. I/O threads receive external messages and place them in work queues that invoker threads service. These threads and work queues implement HALF-SYNC/HALF-ASYNC, which decouples applications from I/O. This allows the system to prioritize its work, which is a key requirement for the overload control techniques that are described later. When an invoker thread passes a message to a state machine, the ensuing transaction runs to completion. State machines only communicate using asynchronous messages, but this creates transient states in a group of collaborating state machines. RUN-TO-COMPLETION CLUSTER addresses this problem by providing priority intraprocessor messages so that a cluster of state machines can reach a stable state before dealing with other inputs.

In a soft real-time system, all threads perform important work, but some threads require more time than others. Priority scheduling is ill equipped to support this requirement. PROPORTIONAL SCHEDULING replaces priorities with factions. A thread's faction depends on the type of work that it performs, and all threads in the same faction receive a minimum percentage of the CPU time.

Extreme systems use distribution to increase their capacity through scalability. Distribution also improves survivability by allowing a component to be replicated. There are many ways to distribute a system's components. HETEROGENEOUS DISTRIBUTION assigns different functions to different processors. HOMOGENEOUS DISTRIBUTION assigns different users to different processors, which is

often simpler and more scalable. HIERARCHICAL DISTRIBUTION combines these approaches, typically so that distribution *across* layers is heterogeneous and distribution *within* layers is homogeneous. Classic SYMMETRIC MULTI-PROCESSING is risky because it reintroduces the widespread need for semaphores. It must therefore use shared memory only for message passing. HALF-OBJECT PLUS PROTOCOL replicates part of an object in another processor to improve capacity.

Extreme systems must avoid software faults. DEFENSIVE CODING means checking pointers, array indices, arguments, and return values. It also means checking incoming messages for errors and using timeouts when sending requests. The most dangerous software faults are those which cause memory corruption. LOW-ORDER PAGE PROTECTION guards against null pointers. STACK OVERFLOW PROTECTION guards against a thread that overruns its stack. USER SPACES firewall applications to prevent them from corrupting each other's data, but this often causes significant performance penalties. WRITE-PROTECTED MEMORY provides read-only access to critical data, eliminating the penalties while still safeguarding against corruption.

When faults occur, an extreme system must detect them and initiate recovery procedures. WATCHDOG monitors a component's sanity with a timer, which the component must periodically reset. If the component fails, the timer expires and the watchdog resets the component. HEARTBEATING is a software implementation of a watchdog. LEAKY BUCKET COUNTER flags an error situation when more than a threshold number of events occur within a defined interval. This prevents the system from overreacting to intermittent faults.

SAFETY NET catches exceptions and signals to prevent a thread from being killed. It then tells the thread to clean up the work that it was performing so that it can remain in service and handle more work. An AUDIT monitors the sanity of key resources. It recovers orphaned resources and fixes corrupt data structures. If a node eventually reaches a point where it must be reinitialized, ESCALATING RESTARTS attempts to return it to service as fast as possible, with a minimal disruption of service. INITIALIZATION FRAMEWORK supports this by structuring the software that `main` invokes to bring the node into service. BINARY DATABASE supports it by reloading configuration data without having to reapply a lengthy sequence of database transactions.

An extreme system is primarily driven by messages, so its messaging system must be both efficient and robust. RELIABLE DELIVERY ensures that internal messages arrive if the destination is reachable and in service. This frees most applications from having to retransmit messages. MESSAGE ATTENUATION avoids message floods by bundling messages or sending them over a period of time.

TLV MESSAGE uses an encoding that is efficient yet flexible enough to support complex protocols with many parameters. Because it encodes parameters individually, it allows PARAMETER TYPE to treat each parameter as a `struct` which a PARAMETER TEMPLATE initializes. If an application writes beyond the end of a parameter, PARAMETER FENCE detects the error. PARAMETER DICTIONARY provides fast access to a message's parameters.

Many messaging techniques reduce copying, which is often the performance bane of message-driven systems. IN-PLACE ENCAPSULATION reserves space at the top of a message buffer so that headers can be prepended as the message travels down the protocol stack. STACK SHORT-CIRCUITING speeds up intraprocessor messaging by bypassing the lower layers of a protocol stack. MESSAGE CASCADING allows an application to send a message to multiple destinations without reconstructing it. MESSAGE RELAYING allows an application to forward an incoming message to the next destination without reconstructing it. ELIMINATING I/O STAGES reduces the number of scheduling and copy operations that occur before an incoming message finally reaches the application software that will process it.

Eliminating copy operations improves performance, but eliminating messages is even better. PREFER PUSH TO PULL achieves this outcome by pushing data to consumers rather than having them use request–response protocols. When responding to a request, NO EMPTY ACKS eliminates bare acks (acknowledgment messages) and replaces nacks (negative acknowledgments) with asynchronous failure messages. When an initial request causes a chain of requests, POLYGON PROTOCOL eliminates intermediate acks by only sending an ack in response to the last request in the chain. CALLBACK replaces a response message with a function call, but its use must be carefully considered because it suffers from a number of drawbacks. SHARED MEMORY eliminates messages by placing data in a memory segment that all applications can access.

When an extreme system receives more work than it can handle, it uses overload control techniques to prevent thrashing and crashing. FINISH WHAT YOU START gives priority to messages that are associated with work that is already in progress. By deferring the handling of new work until progress work is finished, it prevents the system from taking on more work than it can handle. DISCARD NEW WORK throws away new work to ensure that there will be enough resources to handle progress work and to avoid wasting time on stale work that is no longer of interest to clients who have given up waiting for a response. IGNORE BABBLING IDIOTS protects the system from a flood of messages arriving from a misbehaving work source. THROTTLE

NEW WORK hands out credits to work sources to prevent them from sending too many requests to a server.

Extreme systems improve availability by setting aside processors to take on the work of processors that fail as the result of hardware or software faults. Failover techniques reassign work from one processor to another. LOAD SHARING runs active processors in parallel. They share the workload; if one of them fails, the others take on all of the work. COLD STANDBY leaves one or more processors inactive so that they can take over the work of processors that fail. However, this form of failover is slow when an inactive processor must first download its software, and it also drops all sessions. WARM STANDBY overcomes these drawbacks by running a pair of processors in an active-standby configuration. The active processor handles all of the work but maintains loose synchronization with the standby processor, which can then immediately take over if the active processor fails. HOT STANDBY also uses an active–standby configuration but adds custom hardware to maintain tight synchronization between the processors. Although it handles hardware failures more seamlessly, it faces the problem of synchronized insanity when a software failure occurs.

For WARM STANDBY, the primary question is that of how to implement synchronization. Various checkpointing techniques serve this purpose. The first three periodically send updates from the active to the standby processor, typically at the end of a transaction. APPLICATION CHECKPOINTING merely provides a message path between the active and standby processors, leaving all checkpointing details to applications. OBJECT CHECKPOINTING provides a checkpointing framework, but applications still handle many of the details. MEMORY CHECKPOINTING uses hardware to track modified pages, which are then copied from the active to the standby processor. VIRTUAL SYNCHRONY takes a rather different approach. It sends each incoming message to both processors, which therefore run in parallel. This avoids the need for checkpointing, except for when a standby processor enters service and has to catch up with the active processor. In this situation, all checkpointing techniques must use some form of bulk checkpointing to bring the standby processor up to date.

Extreme systems improve availability by installing software without disrupting service. HITLESS PATCHING installs a bug fix in a running processor by inserting new object code for a function and modifying the old function to jump to the new one. ROLLING UPGRADE installs a new software release one processor at a time, so that operators need not take an entire system or network out of service to perform a massive simultaneous upgrade. In a rolling upgrade, PROTOCOL BACKWARD COMPATIBILITY ensures that processors running

different software loads can still communicate. HITLESS UPGRADE uses WARM STANDBY or HOT STANDBY to install a new software release with minimal disruption. It takes a standby processor out of service and loads it with new software and configuration data. The active processor then uses bulk checkpointing to resynchronize the standby processor, after which a forced failover places the standby processor in service so that the same procedure can be repeated to upgrade the previously active processor to the new release. OBJECT REFORMATTING fixes up configuration data and checkpointed objects to conform to their modified layouts in the new software release.

An extreme system provides a number of capabilities that allow craftspeople to engineer its resources, monitor its behavior, and intervene when faults occur. CONFIGURATION PARAMETERS determine the size of resource pools and customize various behaviors. OPERATIONAL MEASUREMENTS provide usage statistics that help craftspeople determine the system's throughput and engineer the size of its resource pools. LOGS inform craftspeople of important events, some of which highlight faults that require manual intervention. ALARMS indicate that the system is in trouble and remain active until the underlying problem has been corrected. MAINTENANCE software detects faults, isolates faulty components from the rest of the system, notifies craftspeople of faults, and attempts to correct or otherwise recover from these faults. Maintenance operations include removing components from service, diagnosing them, reloading or restarting them, and initiating failovers. Maintenance operations may be initiated manually by craftspeople, or automatically by the system itself.

Extreme systems are large, so they often need to disable optional software. CONDITIONAL COMPILATION is often overused but is useful for removing lab-only debug software from production builds. SOFTWARE TARGETING defines an abstraction layer to support different platforms. The abstraction layer provides a common interface (.h file) but selects an implementation (.cpp or .c file) at compile time, based on the target platform. RUN-TIME FLAG uses a boolean to disable optional software, such as field debugging tools, at run time.

Extreme systems must provide tools which allow them to be debugged safely while in service. This rules out large core dumps, writing to the console, and breakpoint debugging. SOFTWARE ERROR LOG and SOFTWARE WARNING LOG use the log system to capture information, such as stack traces, which allow software problems to be analyzed offline. OBJECT DUMP captures the objects associated with a software error log. FLIGHT RECORDER preserves important logs for a *post mortem* analysis after an outage. Truly difficult problems often require online debugging. FUNCTION TRACER captures function calls, and MESSAGE TRACER captures internal and external messages.

TRACEPOINT DEBUGGER captures the contents of data structures at tracepoints rather than breakpoints. All of these trace tools provide triggers that enable them under specific scenarios. This avoids the overhead of capturing everything, which would also cause their buffers to overflow quickly.

As an extreme system grows to support new capabilities, its capacity decreases. Managing capacity is therefore an important activity. SET THE CAPACITY BENCHMARK suggests making your first software release efficient enough to be competitive in the marketplace, but no more. Ideally, it should contain some inefficient code, which you can easily speed up later to offset the capacity degradation caused by new capabilities. For example, many of the techniques that improve capacity for objects and messages should be deferred to subsequent releases. To predict capacity accurately, various profiling tools are required. TRANSACTION PROFILER measures the cost of individual transactions, which are then fed into a model of the system's workload to predict its capacity. FUNCTION PROFILER provides data about

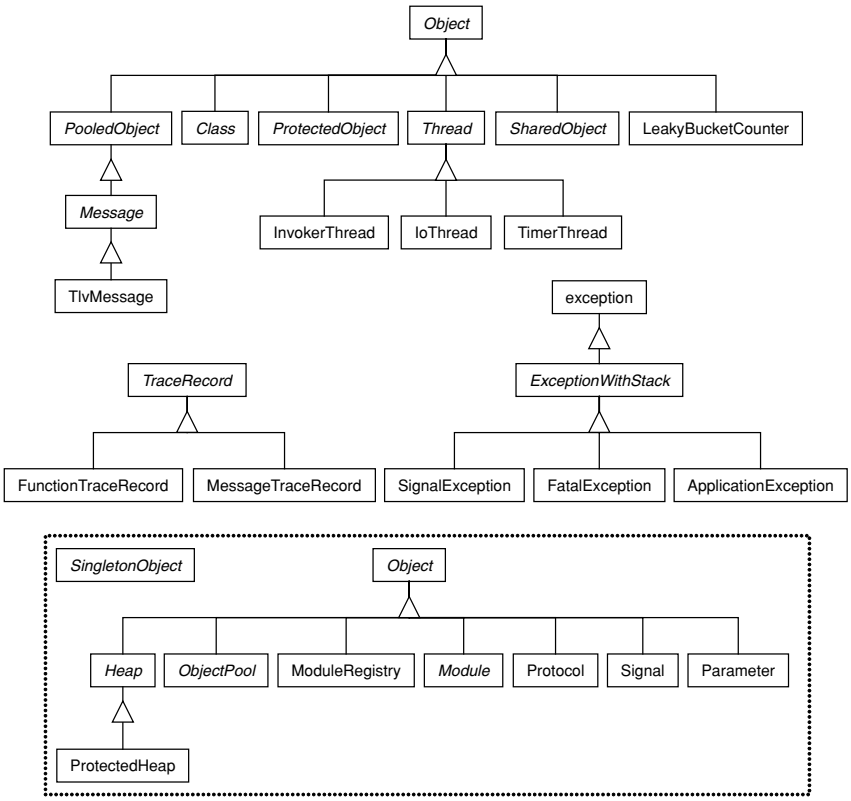

Figure 2.2 Class hierarchy. Most classes derive from Object. Instances of classes in the box are singletons, which use the SingletonObject template.

how often each function is called and how long it runs, which allows designers to focus on speeding up hotspots. THREAD PROFILER provides similar data for each thread.

2.5 CLASS HIERARCHY

To discuss design details, this book introduces various classes. Many of them receive rather cursory coverage, but a few are discussed at length. Figure 2.2 illustrates how all of these classes fit into an overall class hierarchy.

2.6 SUMMARY

- An extreme system contains an administration node, a control node, and service and access nodes.
- A programming model defines techniques that all software components use to achieve desired attributes, such as availability and reliability.
- Because a carrier-grade system must satisfy extreme availability, reliability, capacity, and scalability requirements, its programming model contains many techniques that are not commonly used in the computing industry.