1

Introduction

This book is about programming techniques for constructing carrier-grade software systems. A **carrier** is a firm that operates a communications network which carries subscriber traffic, such as data packets or telephone calls. A **carrier-grade** system is one that meets the requirements of firms which operate these types of networks. These requirements are so strict that this book will refer to carrier-grade systems as **extreme systems**, and to carrier-grade software as **extreme software**. The reason for this is that the requirements faced by carrier-grade systems are starting to show up in other products, such as high-end web servers. Although the programming techniques described in this book have been primarily used in carrier-grade systems, other types of systems can also benefit from their use.

Extreme software is found in products which support mission critical applications. Examples of such products include the routers, switches, servers, and gateways found in communication networks where the underlying transport technology could be internet protocol (IP), asynchronous transfer mode (ATM), synchronous optical network (SONET), or a wireless transport protocol.

This chapter discusses the strict requirements that qualify carrier-grade systems as *extreme* and goes on to describe characteristics that many of these systems share. These requirements and characteristics act as forces that guide the design of the programming techniques which are the primary topic of this book.

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

1.1 REQUIREMENTS FOR EXTREME SYSTEMS

Extreme software systems face rigorous demands in the areas of **availability**, **reliability**, **scalability**, **capacity**, and **productivity**. The following sections describe the challenges presented by each of these forces.

1.1.1 Availability

Users of extreme systems want them to be available all the time: 24 hours a day, 7 days a week, every day of the year. It is therefore unacceptable to regularly shut down such systems for routine maintenance, something that is common practice in information technology (IT) networks. When a telephone switch is out of service, lives are at risk because subscribers cannot make emergency calls. High availability has been a requirement in traditional telephony networks for a long time, but is now infiltrating other networks as users become more reliant on wireless and voice over IP.

Although continuous availability is the goal of an extreme system, it cannot be achieved in practice. Cost, time to market, and the impossibility of testing all possible scenarios preclude it. Nonetheless, the requirements are stringent. The term **five nines** describes almost continuous availability. It means that a system is available 99.999% of the time, which means a maximum of about 5 minutes of downtime per year:

Nines	Downtime per Year	Downtime per Week
2 (99%)	87 hours 36 minutes	1 hour 41 minutes
3 (99.9%)	8 hours 46 seconds	10 minutes 6 seconds
4 (99.99%)	52 minutes 34 seconds	1 minute 1 second
5 (99.999%)	5 minutes 15 seconds	6 seconds
6 (99.9999%)	32 seconds	<1 second

For telephone switches, downtime even includes planned outages required for hardware maintenance and hardware and software upgrades. In the United States, the Federal Communications Commission expects telephone operating companies (telcos) to file incident reports for all outages that last more than 15 seconds. Any noticeable outage in a telephone network typically receives widespread media coverage, damaging the reputation of the telco involved. Such an outage also results in a substantial loss of revenue. These forces, when coupled with the risk of lawsuits if subscribers cannot make

emergency calls, make telcos among the most demanding customers of extreme systems.

In other applications, the requirements may be less stringent. Most customers of an Internet service provider (ISP), for example, would probably be content with three- or four-nines availability. However, large corporate customers, such as on-line stockbrokers, who require more stringent service level agreements, could exclude an ISP providing less than five-nines availability from consideration. Four-nines availability might be acceptable, but only if most outages could be planned for and scheduled during non-business hours.

Achieving five-nines availability in software is difficult. Although many systems claim to be five nines, the fine print often reveals that this only refers to their hardware. In practice, software is far more likely to be the cause of outages in complex systems, which means that a five-nines hardware platform is merely a *starting point* for building an extreme system. In fact, given that the occasional software outage is all but certain, the underlying hardware platform must be *better* than five nines for the system as a whole to achieve five-nines availability. The reason is that the *product* of individual component availabilities determines overall availability when the failure of any component would cause an outage. Thus, if the software provides five-nines availability and the hardware provides six-nines availability, the system provides slightly *less* than five-nines availability ($0.999999 \times 0.99999 = 0.999989$).

Highly available systems typically replicate critical hardware components so that they can survive the loss of a replicated component. Such a loss could still cause a *partial* outage, where *some* users lose service but service continues for *most* users. In other cases, the loss of a component could cause a general service degradation, such as longer response times for all users. The replication of critical components is important because the *system* can then achieve five-nines availability even if its individual *components* do not provide five-nines availability.

1.1.2 Reliability

Users of extreme systems want them to be reliable. Whereas availability refers to a system being in service, reliability refers to it performing correctly. Even if a system is always available, users will deem it inadequate if it is full of bugs which cause it to behave in ways that are not compliant with its specifications. Reliability therefore refers to the overall quality of the system's software.

Although bug-free operation is the goal of an extreme system, it cannot be achieved in practice. Once again, cost, time to market, and the impossibility of testing all possible scenarios preclude it. The question, therefore, is how many bugs are acceptable.

In telephony systems, a typical goal is to mishandle no more than one call in 10,000, which equates to four-nines reliability, or 99.99%. In other systems, such as those that handle financial transactions, the requirements are even more stringent. If a telephone call is mishandled, the user can simply try again. However, if a financial transaction is mishandled, correcting the problem is apt to be much more tedious and expensive, because someone will probably have to correct the error manually.

The term **robustness** describes a combination of availability and reliability. A robust system remains in service and performs correctly even in the face of problems such as hardware failures, illegal arguments to functions, and software exceptions.

1.1.3 Scalability

Extreme systems must be highly scalable. A system is scalable if adding more processors allows it to support more users. The need for scalability arises for two reasons, both of which center around economies of scale:

1. The administrative cost of operating one system that caters to 100,000 users is usually far less than the cost of operating a network of ten systems, each of which caters to 10,000 users. Furthermore, such a network is likely to require closer to 15 systems, not ten, due to the overhead of interconnecting the individual systems into a network.
2. A system has a larger potential market if it is cost effective over a range of configurations. Some customers may want a large system that caters to 200,000 users, whereas other customers may want a small one that caters to 5000 users.

A large telephone switch, for example, hosts at least 100,000 users. During peak calling times, these users might generate 400,000 calls per hour. For each call, the switch receives 11 incoming events:

1. The calling user picks up the phone and receives dial tone.
2–8. The calling user dials a destination address consisting of, say, seven digits.
9. The called user answers.

10. One user disconnects.
11. The other user disconnects.

The switch must therefore handle 4.4 million events per hour – over 1200 events per second. To handle this many events, the switch must be scalable: many processors must share the workload. For example, some processors simply scan the hardware that interfaces to subscriber telephones. These processors detect events such as off-hooks, on-hooks, and the dialling of digits. They then report these events to other processors which handle higher-level details, such as actually connecting calling and called users.

1.1.4 Capacity

Closely related to scalability is capacity: the throughput offered by a single processor. Even if a system is scalable, it will fail in the marketplace if it requires hundreds of processors to do what its competition can do with dozens. Its cost, either per user or per service, will be too high. Thus, a system cannot focus on scalability alone; it must also pay attention to its per-processor capacity.

Customers of extreme systems need to plan the growth of their business. When customers anticipate growth in their user base, they plan to purchase additional systems. However, they become dissatisfied if they need to purchase additional systems because of degradations in capacity, even if scalability can recover the lost capacity. Consequently, customers may insist that a new software release not degrade capacity by more than, say, 3 % when compared to the previous release. This is in marked contrast to the PC marketplace, where new software releases often use so much more CPU time, memory, and disk space that users must upgrade their PCs to reattain a satisfactory level of performance.

1.1.5 Productivity

Extreme software systems are large – sometimes very large. The software for a fully featured telephone switch easily runs to millions, or even tens of millions, of lines of code. Hundreds, or even thousands, of software designers work on such systems. The product lifecycle of these systems spans decades. Most of the telephone switches in use today began their development in the 1970s or early 1980s.

Building extreme systems is therefore costly. Even if a firm develops a product that meets its customers' availability, reliability,

scalability, and capacity requirements, the product's ongoing development and maintenance costs will ultimately determine its success. Designer productivity is therefore an essential element in the design of extreme software. A large number of developers must be able to work on the product in parallel, delivering new capabilities and resolving bugs in a cost-effective manner. Other things being equal, higher designer productivity increases the system's lifespan, and consequently the amount of time over which the initial investment in building it can be recouped.

It has often been observed that, inside a large software system, there is a small system struggling to get out. In other words, much of the size results from artificial, rather than fundamental, complexity. Although many extreme systems could be fairly criticized on these grounds, such criticism also belies an understanding of the forces involved. The requirements, or standards, that specify the behavior of a telephone switch are sufficiently voluminous to fill a large bookcase. A small team of developers cannot implement a system that provides this many capabilities, at least not in a timeframe that satisfies time-to-market requirements. A large development team is therefore required.

A large development team, however, is the primary cause of artificial complexity. To accommodate a large team, a system requires a solid software architecture which incorporates principles such as well-defined interfaces, layering (vertical separation), partitioning (horizontal separation), high cohesion (within components), and loose coupling (between components). The architecture must be well documented, and all software developers require training in its use. If the architecture is inadequate or poorly understood, the lack of an overall vision leads to a system with multiple, conflicting architectures. Adding new capabilities becomes difficult because of the need to fit into these conflicting architectures. Eventually the system deteriorates to the point where developers cannot predict the consequences of their changes, and so bug fixes often replace old bugs with new ones. When a system reaches this point, it must be rewritten – at a substantial cost, and at a high risk of missing its market window.

The software architecture of an extreme system must make it easy to add a new capability – such as a new hardware device, service, or protocol – without churning existing software. This reduces the risk of introducing bugs in proven code and allows far more developers to work in parallel, because they do not get in each other's way. Layering and partitioning are the primary ways of achieving these outcomes. They make it easier to add new capabilities because developers do not have to understand the entire system, only the parts in which they are working. Layering and partitioning also allow

new products to deliver capabilities independently while sharing a common platform.

The software architecture should also make it possible to add a new capability without cloning and tweaking existing software. And when a new capability is not required by all customers, it should be possible to make it optional without resorting to the use of conditional compilation (`#ifdef`, for example). These regrettably common techniques, of clone-and-tweak, and the indiscriminate use of conditional compilation, lead to unmaintainable and incomprehensible code.

Because extreme systems are large and complex, they need special debugging tools. Common techniques such as setting breakpoints and writing information to the console fail to pinpoint problems quickly and cannot be used in live systems. To improve productivity and support debugging in the field, extreme systems must implement comprehensive trace tools and software logs.

1.2 BECOMING CARRIER GRADE

When a system faces extreme availability, reliability, scalability, and capacity requirements, its software must often use techniques that are not common practice in the computing industry. Some of these techniques do not necessarily apply to systems in which one or more of these requirements is absent. For example, avionics software for an aircraft flight control system must be highly reliable. It must also be highly available – but only when the aircraft is actually in use. However, it need not be highly scalable, because it only serves *one* aircraft. In contrast, the software for an air traffic control system faces all three requirements. Not only must it be reliable, it must be available whenever aircraft are flying, and it must be scalable to the point where it can track thousands of aircraft simultaneously.

The primary focus of this book is software design. This is probably the most neglected facet in the design of carrier-grade systems. Few books deal with it in a comprehensive manner. This book attempts to fill that void.

A system requires continuous improvement to become carrier grade. It does not happen immediately. Carrier gradeness is as much a journey as a destination. A good design can eventually become carrier grade. A poor design, however, will *never* become carrier grade. If you are developing a carrier-grade system, you need to be aware of the techniques described in this book. However, it is unlikely that you will use most of them in your first software release. Time-to-market pressures simply preclude this from happening. Towards the end of

the book, we will therefore revisit each technique by placing it into one of these categories:

- must be implemented in your first software release;
- can be implemented in a subsequent release if your initial software design makes provisions for its eventual inclusion;
- can be implemented in a subsequent release without prior planning.

Satisfying all carrier-grade attributes – availability, reliability, scalability, capacity, and productivity – is challenging, partly because they often conflict with each other. For example, reference counting (tracking the number of references to an object so that you can delete it when all references disappear) improves reliability but degrades capacity. Application frameworks improve productivity but also degrade capacity.

A system cannot become carrier grade as the result of its software alone. All of the elements that contribute to the system must be carrier grade. The system requires carrier-grade hardware. It must be designed using a solid development process which includes rigorous stress testing. Before it is deployed, it must be modelled and engineered so that it will survive times of peak usage, even when presented with more work than it can handle. And when it is deployed, it must be easy to operate and thoroughly documented. Carrier-grade systems sometimes suffer outages as the result of procedural errors – human errors made by the craftspeople who monitor and operate these systems. When these outages result from poor documentation or overly complicated operational procedures, customers will rightly attribute them to the provider of the system rather than to the craftspeople. It is beyond the scope of this book to cover all of these topics. Fortunately, many publications discuss development processes, hardware design, system engineering (quality of service guarantees, for example), documentation, and operability.

Building a carrier-grade system is challenging and costly. You therefore need to decide, early in your product's lifecycle, whether it eventually needs to become carrier grade. If the answer is yes, you must plan its evolution accordingly and guard against being distracted from your path. Getting distracted is easy because carrier gradeness is usually important only in mature products. For a new product, content usually determines success. At first, customers primarily want you to deliver new capabilities quickly. Later, as the product and the customers' businesses mature, customers focus more on predictability. They want you to deliver software releases on time, at predictable intervals, so that they can plan the deployment of new capabilities. Carrier gradeness only becomes a customer focus later,

when the product is commoditized. At this point, carrier gradeness is a differentiator, but you will have little chance to attain it unless you planned for it from the outset. Unfortunately, there have been many cases where an innovative product initially enjoyed success, only to have its lifecycle cut short when its market matured and competitors displaced it because it could not meet its customers' quality expectations.

1.3 CHARACTERISTICS OF EXTREME SYSTEMS

This section defines characteristics of extreme systems. Most extreme systems are **embedded**, **reactive**, **stateful**, **real-time**, and **distributed**. If a system does not exhibit all of these characteristics, some of the techniques that we discuss may not be the most appropriate or optimal for it.

1.3.1 Embedded

An embedded system focuses on a specific application, such as routing packets or handling telephone calls, and it often contains custom hardware developed for that application. Other applications cannot interfere with embedded applications, which is why extreme systems are usually embedded systems.

Because an embedded system is dedicated to a specific application, it runs the same software for a long time. Consequently, its designers can study its behavior before deploying it. They can build models to predict its behavior and can tune it for optimal throughput.

Because an embedded application runs alone, you can customize its operating system. Granted, this will decrease its portability, but if its platform also contains custom hardware, it is not that portable in the first place. Moreover, modifying the operating system might improve reliability or capacity. Certain characteristics of commercially available operating systems are not well suited to extreme embedded systems. We will look at these characteristics and discuss ways to modify or avoid them.

1.3.2 Reactive

A reactive system responds to external inputs. The inputs are usually messages that arrive from users or, more generally, various devices. Applications receive messages and react appropriately. When an application performs work, it often sends messages itself. Some of these

are destined for users or devices, but others are destined for other applications within the system.

Each message belongs to a **protocol**, which defines

- a set of message types, which this book refers to as **signals**;
- a set of **parameters** – additional information that accompanies the signals;
- the order in which signals may be legally sent and received, and
- for each signal, whether a parameter is mandatory, optional, or illegal.

A **message** consists of a signal and zero or more parameters.

Many reactive systems lack a proper description of the protocols that drive them. If you look at their code, you can only discern the protocols by tediously studying the applications, which are themselves implemented in *ad hoc* ways. To provide traceability between their requirements and their software, reactive systems need to include protocols as key elements of their object model.

The term **transaction** refers to the work that a reactive system performs when it receives a message. When a message arrives, the system places it in a work queue. Later on, an application processes the message by performing some work, which may also involve sending some messages. When these messages are internal (that is, destined for other applications within the system), they result in additional transactions.

1.3.3 Stateful

A stateful system must remember a user's or device's current state so that it can handle subsequent events correctly. When a subscriber picks up a phone, for example, a telephone switch must know whether it is currently offering a call to the subscriber. Only in this way can it know whether to treat the off-hook event as an answer or as a request to set up a new call.

This book uses the term **session processing** to describe the behavior of stateful systems. A **session** consists of a series of transactions that are related by state. For example, a telephone call, from start to finish, comprises a single session. To be somewhat more accurate, it actually comprises two sessions: one for the calling user, and one for the called user. Each session consists of a sequence of transactions. For the calling user, the transactions are going off-hook and receiving dial tone, dialling digits and receiving ringback tone, receiving answer from the called user and connecting the speech path, and

disconnecting. For the called user, the transactions are receiving a call and applying power ringing, going off-hook and connecting the speech path, and disconnecting. To handle each transaction correctly, a session must maintain state information.

In contrast to session processing, **transaction processing** refers to the handling of transactions in *stateless* systems. Systems that process financial transactions are a good example of this. They are stateless in the sense that they handle each incoming, external message (that is, messages from outside the system) atomically. A transaction may access and update a database, but it is otherwise stateless. In other words, the system does not contain sessions that are waiting for additional external inputs.

In this book, we will focus on stateful systems (session processing) rather than stateless systems (transaction processing). There are various reasons for this. First, stateless systems can be viewed as a subset of stateful systems. Second, stateful systems are generally more complex than stateless systems. Finally, many stateless systems face a different balance of forces than do stateful systems. For example, reliability is more important in financial systems than it is in communications systems, but the opposite is usually true for availability. Financial systems therefore use patterns like Two-Phase Commit [GRAY93, BERN97] to improve their reliability. This pattern, and others that apply to reliable database systems, are not covered here because they are discussed in other books and are of limited use in stateful systems.

Although our focus is on stateful systems, many of the techniques that we cover also apply to stateless systems. The reason for this is that, although a system may appear to be stateless from a black-box perspective, it is often stateful from a white-box perspective. A system that processes a financial transaction, for example, may distribute its work among many processors. An incoming, external message therefore creates a session to coordinate the sending and receiving of the internal messages that implement the transaction.

Many stateful systems lack a proper description of the state machines which underlie them. If you look at their code, you can only discern the state machines by tracing through different sequences of remote procedure calls. To provide traceability between their requirements and their software, stateful systems need to include state machines as key elements of their object model.

What a reactive, stateful system truly needs is a session processing framework which defines base classes for implementing protocols and state machines. Such a framework can dramatically improve both productivity and quality. Internally, it will embody many of the techniques described in this book. However, its overall design

is a topic for another book. Until such a book becomes available, see [SGW94] and [UTAS01]. The first of these addresses the topic primarily from a modeling perspective, and the second addresses it from the perspective of a specific application domain.

1.3.4 Real-Time

A real-time system needs to perform work before a deadline occurs. There are two general types of real-time systems, based upon the strictness of their deadlines:

- A **hard real-time** system is one that *must* perform work before a deadline. If it fails to do so, it will perform incorrectly, perhaps with serious consequences.
- A **soft real-time** system is one that *should* perform work before a deadline. If it fails to do so, the consequences will be less serious, but its users will nonetheless be dissatisfied with its response time.

Most extreme systems face both hard and soft real-time requirements. Hard real-time requirements usually occur in low-level software, such as device drivers and other software which interfaces directly with hardware. Soft real-time requirements usually occur in higher-level software that provides services to end-users.

Most books on real-time systems discuss techniques which apply to hard real-time systems. Some of these techniques are appropriate for soft real-time systems, but others are not. Our primary focus will be soft real-time systems, so a similar warning applies: not all of the techniques in this book are appropriate for hard real-time systems.

1.3.5 Distributed

A distributed system partitions work among many processors for a combination of the following reasons:

- To provide scalability when a single processor cannot keep up with all of the work. Here, the purpose of distribution is to offload work through delegation.
- To improve availability when another processor takes over the work of a failed processor. Here, the purpose of distribution is to improve survivability.

- To simplify its design by separating software with hard real-time requirements from software with soft real-time requirements. The reason for this type of separation is discussed in Section 5.4.3.

Distribution often simplifies local (per-processor) complexity by restricting the types of work that a processor performs. At the same time, however, it increases the system's overall complexity. The first problem is how to partition the work. Another is how to handle partial failures. We will look at these issues and others in Chapter 6.

1.4 SUMMARY

- Carrier-grade systems must satisfy extreme availability, reliability, capacity, and scalability requirements. Consequently, this book often refers to them as *extreme systems*.
- Carrier-grade systems are usually embedded, reactive, stateful, real-time, and distributed in nature.
- It takes time for a system to become carrier grade. It must include certain carrier-grade techniques in its initial software release, but it will not have enough time to include all of them. It therefore needs a plan for incorporating other techniques in subsequent releases.