

Concurrency in Computer Applications

*Software Designed for Achieving Concurrency in Computer Applications for Monitoring,
Management and Control*

David L Nelson

Contents

Contents	2
1 A Brief History of Concurrency	7
2 Recovering Concurrency	11
3 The Savoy Server	15
3.1 State Diagrams – Importing and Exporting State Transitions	16
4 Domains	19
4.1 Finite State Models	20
4.2 Achieving Concurrency in Domains	21
4.3 Normalized Interface	22
4.4 Some Generic domains	24
5 Rule Engine: Integrating domain Activity	29
5.1 Binding	31
5.2 Completing the Rule Grammar	32
5.3 Forward Chaining	33
5.4 Rule-based Procedures	33
5.5 Comparison to Threads	35
5.6 Bounded Buffer Problem	35
5.7 Discussion of Rule based Languages	36
5.8 Consistency	37
5.9 Implementation	37
6 Distributed Computing	41
6.1 Network Architecture	41
6.2 Client-Server Peer-Peer Symmetry	42

<i>CONTENTS</i>	3
6.3 Managing Network Complexity	42
6.4 Computer to Computer Communications	45
6.5 Summary	52
7 Client Applications	55
7.1 Stateless Clients	55
7.2 Thin Clients versus Stateless Clients	56
7.3 Components of the User Interface	57
8 Putting it all Together	61
Index	63

Concurrency is the perceived ability of a computer system to perform many tasks in parallel. It has been at the center of considerable development in the computer industry since its inception, and continues to this day. Why is concurrency so important? The answer is simple: in the real world, there's a whole lot going on, and it's all happening concurrently. To build a computer application that effectively deals with the massively concurrent real world, one must go to extraordinary lengths to avoid destroying the concurrency required to manage it all. This book presents an integration of three methods all directed at achieving concurrency at the application level: (1) message passing, (2) rule processing, and (3) name scoping. It further describes the implementation of a specific product developed by Savoy WebEngines, Inc. and, in doing so, serves to document much of its internal architecture as well. The presentation follows the layering, bottom to top, of the system hopefully conveying the key features that contribute to concurrency. Message passing has been used in operating system design since the early 1970s, recently bearing the acronym MOM for Message Oriented Middleware. A recent book, *ENTERPRISE INTEGRATION PATTERNS* by Hohpe and Woolf*, provides a thorough description of messaging together with descriptions of commercially available products. Their focus is the integration of multiple applications at the enterprise level. While this book focuses on concurrency within an application, much of their work can be applied here as well.

The most apparent application of messaging occurs in operating systems like Windows where the user environment is flooded with short messages to maintain concurrency on your desktop, performing text updates, graphic rendering, animation, sound, and more. In other areas, message passing has not received the attention it deserves – it is not formally used in networking, for example, nor in client-server applications. We introduce the concept of a 'message thread' which is an independent process of a message circuit performing a potentially complex exchange of information. The deployment of message passing technique across distributed servers and up into client applications is a central focus of this book.

The second method is the use of a rule-based language for expressing the interactions of message threads as well as finite-state systems. A particularly novel aspect of this is the full integration of rules and messages forming a high performance coherent architecture for achieving highly concurrent applications. A highly optimized rule engine, in effect, performs fast switching across a complex of message threads.

The third method is name scoping which is a general mechanism of conditioning messages on a 'need-to-know' basis so that in principle, no information is delivered that is unneeded. Scoping could be compared to sub netting which we regard as a

*Enterprise Integration Patterns, Designing, Building, and Deploying Message Solutions, Gregor Hohpe and Bobby Woolf, Addison Wesley, 2004

rudimentary way of controlling the propagation of messages.

Absent any formal approaches, software developers are somewhat encouraged to destroy concurrency. The simplest case is where a program requests input from a user, suspending the application until the user responds. This trivial case is so apparent that no one actually designs applications that operate like this, but it typifies so-called synchronous logic that causes programs to suspend during I/O, database queries, network RPC (Remote Procedure Call) calls, and more. There are many programming techniques to deal with these problems such as running multiple threads within an application, but none of them are well supported by languages, and the tools available are subject to extraordinary misuse which often leads to unreliable designs.

I began developing computer systems in the 1960s for the control of particle accelerators used in experimental physics and have been actively developing software during the 45 years since, always involved in one way or another with the complexities of dealing with real-world environments. Out of this background has come the realization that the standard platforms that the industry has provided do not lend themselves toward building highly concurrent applications, and consequently I have developed a number of design concepts that facilitate concurrency. Later chapters will discuss advantages of asynchronous rule processing, client/server threading, message based clients, and more.

Numerous texts[†] have been published that explain the design and use of techniques used to build concurrent applications, such as multi threading, mutual exclusion, semaphores, message passing, and more, and consequently this book does not dwell on them in detail and the reader is referred to them when appropriate. Nor is this about parallel computer architecture. This book is rather a practical description of techniques which I have found to work for me. It describes the application of these concepts in the design of a state-of-art system for monitoring and control of large distributed facilities. With few exceptions the designs have been implemented under both the Windows and Linux operating system and have been deployed in real commercial applications. In many cases, these products have been operating in the field under stressful conditions without errors, without crashes, 24 hours a day non-stop since 1998 - more than a decade of continuous operation.

Using a properly designed concurrent application is an unmistakable experience. Multiple components that comprise the user interface appear to all be working all the time. Alerts and notifications from subsystems and remote facilities are properly managed. The system automatically adjusts to increasing demands, bringing

[†]CONCURRENCY: STATE MODELS AND JAVA PROGRAMS, Jeff Magee and Jeff Kramer, Wiley and Sons

additional remote facilities online automatically and in a nearly seamless fashion. Network bandwidth is automatically managed and as the latency grows due to busy networks, the application components accommodate it. User dialogs requesting information patiently wait for input while the background continues to work as expected. This is not your father's Browser-based web application, but rather a system that reflects the liveliness and concurrency of the real world to which it is connected.

This second edition has been modified from the original 2007 version principally in the discussions of Linux. Since that time, the server has been totally re-implemented under C# using .NET libraries and a very impressive Open Source project called Mono [‡] has permitted it to run on both Linux and Mac computers. The Mono runtime system has removed all of my previous concerns regarding Linux and has provided a powerful framework that in every regard is as powerful as that found under Windows. Consequently, the entire discussion of operating systems and their differences has been dropped.

The second edition also includes Chapter 3, a more general discussion of the motivation behind the design of the Savoy server. Having little to do with concurrency per se, and more to do with software complexity and how to build scalable applications, this chapter describes a particular approach to designing software systems that we have found to be important and useful.

Framingham, MA
March, 2009

[‡]www.mono-project.com, a cross platform environment sponsored by Novell

Chapter 1

A Brief History of Concurrency

At the dawn of the development of digital computers there ensued a debate as to the relative merits of solving problems by programming a digital computer versus using a well established technique of building analog computers. Analog computers, so the arguments went, were faster, often operated in real time, and were intrinsically concurrent, solving complex non linear mathematics through the use of metaphoric components like capacitors, inductors, resistors and the like. The principle advantages of going digital were accuracy and flexibility, but at a significant cost - you had to transform the problem into a sequence of single steps, involving iteration, approximation, perturbation, etc. - simply because you were solving a parallel application on a sequential computer.

But it wasn't always that way. Early computers had a high degree of concurrency, attacking their analog counterparts head on. The ENIAC* (1946) was the first all-electronic digital computer developed at the Moore School of the University of Pennsylvania. While other computer developments at that time used electromechanical techniques (relays), the ENIAC was constructed from 17,468 vacuum tubes, borrowing heavily on the 1937 era work of Atanasoff, an Iowa State physicist who first conceived of the idea of using electronic circuits to compute digital logic. The then massive ENIAC had 20 independent arithmetic processors all of which could operate concurrently. But all of this concurrency came at a high price; the setup time for a new problem was dramatically longer than the electromechanical competitors of the time, notably the Harvard Mark I.

The problem that the ENIAC faced was where to get its instructions. Contemporary machines at the time read instructions from card readers which ran at about the same speed as the relay logic of their processor. But with ENIAC, this wouldn't

*THE COMPUTER FROM PASCAL TO VON NEUMANN by Herman Goldstine, pg 255

do, since the card reader would be way too slow, and so, the instructions had to be manually entered with patch cables which could subsequently be read by the high speed circuits. Then there was the issue of keeping the large number of accumulators busy since they could all operate concurrently. It was not unusual for a thirty minute problem to be comprised of 28 minutes of setup and only two minutes of computation.

Nearly a year before the ENIAC was completed, there occurred parallel developments in the U.S. and in the Britain which would have profound effects on the development of computing for the rest of the century. It could be argued that all of the developments since the year 1945 are mere footnotes to the insights of both the brilliant mathematician John von Neumann working in the U.S., and those of Alan Turing working in Britain, and while von Neumann has received much of the credit, in many respect, Turing's work was more significant. In the US., during the year before the ENIAC was finished, the designers began working on a successor machine called the EDVAC (completed in 1950[†]) which was intended to incorporate many new ideas that would simplify the 'programming' of the machine. Although it was a collaborative effort, much of the credit has gone to von Neumann who wrote the initial report of the EDVAC. This report was never intended to be a publication and so it lacked due credits to other team members (notably Mauchly and Eckert), but published it was, and so von Neumann's name has become exclusively associated with what is now called the von Neumann architecture, as well as the pejorative reference, the von Neumann bottleneck.

The 'bottleneck' refers to the single point of connection among the components of the EDVAC, such as an arithmetic processor, a memory system, an input/output system, and so on. The most significant aspect of this was the idea of consolidating the storage of internal data with external instructions into a single common memory system - what we have since referred to as a 'stored program'. This was uniquely required for high performance machines in order to avoid excessive waiting for instructions from mechanical devices like card readers. Storing both programs and data into a common memory was considered to be a revolutionary development. Second in importance was the replacement of a collection of arithmetic processors by a single processor that would operate one instruction at a time. This was less important at the time because a machine that ran at the speed of light could easily

[†]EDVAC was preceded by the EDSAC (Electronic Delay Storage Automatic Calculator) built at the University of Cambridge by Maurice Wilkes, who first read the EDVAC Report in May 1946 (given to him by Leslie Comrie); EDASAC, operational in June 1949, incorporated many concepts published in the EDVAC Report including stored program, serial execution, binary arithmetic and more.

absorb the performance hit if it resulted in greater flexibility and simplicity. So, the von Neumann architecture had the costly consequence of reducing the parallel architecture of the ENIAC to the sequential architecture of the EDVAC where operations were performed serially, one at a time. They purposefully sacrificed concurrency to achieve simplicity. One of the reasons they could afford this tradeoff was due to the incredible speed advantage that the ENIAC's electronic circuits (100KHz cycle time) had over the electromechanical circuits of their chief competitor, the Harvard Mark I. von Neumann's insight paved the way for the development of computer languages, but at the cost of removing concurrency from computers for many generations to come.

Meanwhile, in Britain, there was mathematician Alan Turing, an unsung hero in this area who has been relegated as the father of artificial intelligence (AI). This attribution falls short of acknowledging his more significant contributions to modern computer architecture. Nearly ten years earlier (1936) Turing had published his seminal paper on 'Computable Numbers' where he laid out the organization of a machine that bears his name, the 'Turing Machine', having a processor that read and wrote information to and from a memory system, envisioned as an infinite tape. Turing's paper addressed a particular mathematical problem laid down by Hilbert, who, in 1900, posed twenty three unsolved problems to the mathematical world involving completeness, consistency and decidability of mathematical statements. Hilbert's notion of decidability involved 'sequences' or steps that could be encountered in a proof which in turn suggested the notion of a 'cycle' in Turing's machine. In his 1936 publication, Turing, perhaps unwittingly, happened to place his 'instructions' on the same 'tape' as his 'data' and in so doing created the von Neumann architecture a decade ahead of its time.

It was nearly ten years later, however, in a February 1947 talk where he presented the concept of hierarchical subroutines, the separation of programming from machine development and developing software to aid the development process:

"This process of constructing instruction tables [procedures] should be very fascinating. There need be no real danger of it ever becoming a drudge, for any processes that are quite mechanical may be turned over to the machine itself." – Alan Turing, 1947[‡]

In fairness, it should be noted that these ideas were also set forth by von Neumann and Goldstine in their paper Planning and Coding[§] dated April, 1947 The net

[‡]Alan Turing, the Enigma, by Andrew Hodges, pg 326.

[§]The Computer from Pascal to von Neumann by Herman Goldstine, pg 255

result of all of these developments was immediate agreement regarding the overall organization of future computers. There shall be a central processing unit executing a stored program of instructions from a common memory - one instruction at a time.

Chapter 2

Recovering Concurrency

Since we're restricted to the sequential execution of a single machine, we can create the perception of concurrency by rapidly switching (interrupting) what this machine does among multiple tasks so that all tasks appear to run concurrently. With programs and data both resident in a common high speed memory, it became necessary to move information in and out of this memory as rapidly as possible and this was solved by early developments of magnetic tape systems. The first computer to incorporate interrupts for the purpose of achieving concurrent operation (in this case overlapping computation with input/output) was the IBM's Naval Ordnance Research Calculator (NORC) which was operational from 1954 to 1968.

"The NORC also included the first input-output channel, which synchronized the flow of data into and out of the computer while computation was in progress, relieving the central processor of that task, a concept that was quickly adopted across the industry."*

Interrupts became a common facility of computers, typically tied to I/O devices and a real time clock.

The first attempts to recover higher levels of concurrency occurred in the development of multi programmed operating systems, notably the MIT Compatible Time-Sharing System (CTSS) project (1962), which multiplexed a sequential computer across what appeared to be separate independent programs. Concurrent tasks are commonplace these days - every PC has them. But when first developed, it was a bit counter intuitive. I recall in the mid 1960s observing, with some astonishment, the output of two applications (decks of punched cards) appear on the printer in

*<http://www.columbia.edu/acis/history/norc.html>

reverse order to their submission. The second application overtook the first because it ran faster – a novelty at the time, since most systems operated in 'batch' mode which preserved their order from start to finish. Of course, we knew technically what was happening, but the first experience of it was notable.

Concurrency necessarily involves interaction, and these issues were largely addressed by E. W. Dijkstra, C. A. R. Hoare, and Per Brinch Hansen during the late 1960s through the 1970s. These researchers invented techniques for managing the interaction of concurrent programs, including Semaphores, Monitors, and the like.

The first attempt to formalize the problem of process synchronization was proposed by Dijkstra which became known as 'p,v' operators, or semaphores. Semaphores permitted two processes to synchronize with each other with symmetry as to which one got there first. Dijkstra also proposed methods of avoiding deadlocks in software design that can cause systems to hang while waiting for unattainable resources.

Still, while semaphores were an effective mechanism, their use could be enhanced by implementing them in higher level data structures that became known as Monitors, which controlled the access to shared variables within global objects.

Around the same time, operating systems were designed using message passing schemes that permitted concurrency, frequently bypassing problems posed by concurrent processes. In message passing systems, many concurrent segments of code run non-preemptively in the same process. Microsoft's early DOS operating system slowly evolved toward a message passing approach following the earlier success of the Macintosh. So, while Mac OS and early Windows were developed just a few years after UNIX, their differences on how concurrency is managed carries forward to this day, with Windows implementing messaging within single large processes and UNIX implementing more traditional approaches across many, smaller processes.

The 1980s saw the rapid development of Local Area Networking (LAN) and with it numerous schemes to exploit the concurrency afforded by parallel networked computers. Remote Procedure Call (RPC) techniques introduced to UNIX by Sun Microsystems facilitated means whereby a process on one computer could invoke a procedure on another. Later, Apollo Computer created the Network Computing System (NCS) which introduced the role of a broker in establishing dynamic linkage for RPC, and this later evolved into what is referred to as the CORBA standard now in use by many vendors.

As the issues surrounding the choices for distributing applications across networks matured, they can now generally be cast into two approaches:

- RPC (Remote Procedure Call), a synchronous method of requesting remote service execution requiring consumers to suspend service execution until they receive a reply from the service provider, or

- MOM (Message Oriented Middleware), an asynchronous messaging passing service where the consumers of the message are physically and temporally decoupled from the service providers.

The two schemes have been extensively characterized and compared, however for reasons that remain unclear, message passing has not become as formalized and standardized in either peered networking, nor in client/server networks to the degree that its RPC alternative has.

Chapter 3

The Savoy Server

One purpose behind this writing is to document the architectural features of the Savoy Server, a commercial product of Savoy WebEngines, inc. Concurrency in the Savoy product is achieved through very efficient message passing techniques. However, the principle motivation behind the design of the Savoy system was not concurrency, but rather from observations regarding how best to organize complex software applications.

The design of the Savoy Server was motivated by a vague reference to Occam's razor, which states that when several acceptable explanations for a phenomenon exists, the simplest is preferable. Simplicity as a design goal is certainly not new and every software engineer is encouraged to 'keep it simple'. Still, software applications are generally quite complex.

Contrast this to many of the most profound scientific discoveries that have simple representations. One of the most significant scientific discoveries of all time is Newton's law of gravitation which is expressed by the simple equation

$$F = \frac{Gm_1m_2}{r^2} \quad (3.1)$$

Similarly, Einstein's famous Theory of Relativity is generated by the equation

$$E = mc^2 \quad (3.2)$$

and his General Theory of gravity is

$$G_{uv} = \frac{8\pi G}{c^4} T_{uv} \quad (3.3)$$

Further, Feynman's theory of quantum electrodynamics can be represented by his relatively simple Feynman diagrams. The result of decades of research into particle

physics is represented by what is called the Standard Model, which embraces a simple concept of symmetry to describe an organized array of elementary particles.

This view of scientific principles is the result of what is termed 'reductionism' and the question we raise here is whether we can apply similar principles to software design. That is, can we design complex software so that it deals exclusively with simple representations? We derive hope for this question by observing that many descriptions of the most complex software functions can be reduced to a simple phrase. For example, a function of a video surveillance system could be described by *If a person walks up to the front door, ring a chime*. Another example might be *If an on-line user wants to purchase an item, move a description of it to a shopping cart*. A further example is *If a video system detects an intruder in your home, sound an alarm and call the police*. Finally, *If a scanned finger print matches that of a known person, open the door*.

Each of these are expressed by a simple IF-THEN-ELSE rule, the components of which could be extremely complex. Scanning a finger print may be complex as is perhaps opening a door. But the relationship between these functions is simple Boolean logic. The question we turn to then, is how do we cast complex software modules into a form whereby simple phrases can both describe and control their behavior?

3.1 State Diagrams – Importing and Exporting State Transitions

State diagrams have been used for many years to describe and analyze the behavior of complex systems. The interesting aspect of state diagrams is that the phrases we described above can generally be correlated with a simple state transition in a state diagram. Consequently, we entertain the notion of importing and exporting certain state transitions.

State transitions which are imported or exported are referred to as 'assertions' within the Savoy server. So, an assertion to the software module 'asserts' a specific transition within the state diagram. Similarly, whenever certain transitions occur within the module (due to external influences, for example), the module 'asserts' a simple description of that transition.

Clearly, then, assertions into and out of the module can both control the module and express its behavior. And, the descriptions of these assertions can become components in the Boolean logic previously described. Since simple rule logic can combine assertions across multiple modules, this becomes a way of coordinating the

complex behavior of a collection of modules, thereby forming a complete application.

The Savoy Server is organized as sets of cooperating modules, called Domains, having the simplest of relationships with other Domains, coordinated by a Rule Engine. Domains are generally associated with device hardware or complex subsystems, like Internet Sites, data bases, communication facilities, and so on.

Chapter 4

Domains

We will proceed through the design issues of a complete system hierarchically, beginning at the bottom, that is closest to the hardware, and working our way upward through various levels to the user interface, addressing the issues of concurrency at each level.

At the lowest level of the system we are describing are 'domains' – software modules that interface to real world facilities. Domains are somewhat abstract and they are intended to present normalized interfaces to higher levels of the system so that all domains can be treated similarly. Domains are most frequently associated with a hardware interface to data acquisition or control subsystems. However, domains can be much more than that, and include such things as websites, remote computer systems, or other subsystems operating within the computer. A running system might be concurrently running dozens of domains and we shall treat the problem of integrating their activity in the next chapter. But given the possibility of operating a large number of domains in the same system makes it pretty clear that the logic of any single domain had better not permeate to higher levels in the system, as that would add a considerable amount of domain-specific software support.

Consequently, we must normalize our domains. Normalization doesn't necessarily mean to make them all the same. Many systems, like LONWorks, for example, attempt to put (say) all domains of a similar nature into a common architecture, spelling out in detail the properties, permissible values, units of measurement,, and so on. This considerable task seems rudimentary and in the long run, it limits the capabilities of the domain. I prefer to simply normalize the interface to the domain, not its capabilities.

Interfaces to software components (like OLE 2.0 used in Microsoft's ActiveX architecture) has been the subject of extensive development over the years, and

have included features such as versioning to protect clients of the component against changes in the interface. These techniques have always seemed overly complex to me, and as such, I have adopted a different approach to the software interface and that is to adopt an interface that is constant across all domains for all times. This may seem like a challenging problem, but in fact it is most trivial.

While most interfaces follow some application specific procedure call, accompanied by a proper set of calling parameters, such as 'Procedure SomeFunction(type1 var1, type2 var2)', the way to make the interface domain independent is to route all function through a common portal function with a very limited set of parameters. The parameters should be no more than (1) a short integer for control purposes, a text string that can be interpreted quite generally, and an optional binary payload to be used according to the control function.

In principle, all functions could be routed through this common portal, though it may be efficient to create a few portals as long as their purpose is generic to all conceivable domains. We will discuss a good set of portals in the next section, but first a word about the pros and cons of common portals. The advantage of application specific procedures is that the calling parameters are type-checked at compile time so they don't have to be checked at runtime. Accordingly, a disadvantage of a common portal, particularly one with a text string as a calling parameter, is that the string must be parsed at runtime in order to defend itself against badly formed string data. This issue is considerably alleviated, however, by restricting the author of the string to a companion component in the system that 'promises' never to create a text string that the domain can't parse. This companion component could be part of a user GUI system that creates the logic for the domain (in our case a Rule Editor discussed later), and as long as the companion component is developed as a partner with the domain component, and as long as the system assures that no erroneous text string will ever be delivered to the domain, we have, in fact, facilitated a pretty good form of type-checking, arguably superior to that provided by standard compilers. We will cover this form of type checking in a later chapter on rule formation.

4.1 Finite State Models

Most of the domains associated with complex acquisition and control systems can be cast into what is referred to as a finite-state system. There are two aspects of finite-state systems that are important here: first, that, as the term implies, subsystems having continuous state do not readily comply (more on this below), and secondly, that all activity of the domain can be modeled as a 'change of state'.

Continuous versus Discrete Subsystems

Continuous systems are those whose parameters vary continuously, like temperature, pressure, voltage and so on. Conversely, discrete systems have denumerable state, like 'off/on', 'true/false', '1/2/3/4', and so on. In many cases, continuous systems can be quantized into discrete systems by casting the parameter into discrete values, like temperature as 55,56.92, etc.. Moreover, continuous systems can be cast into discrete values by applying 'fuzzy logic' so that temperature becomes 'cold/cool/nice/warm/hot' for example. This conversion is intended to be totally encapsulated within the domain, so that in general, the domain interface deals exclusively with the finite state.

Fuzzy logic can take on many interesting forms. For example, in Video compression domains data is acquired from cameras at a high rate and transferred up the binary interface bypassing any finite state logic, as it would seem unreasonable to cast a camera's pixel value as a change of state. However, by comparing pixel values against discriminator limits over defined regions of a video image, we can translate changes in the image (due to motion, for example) into a change of state and run that occurrence through the finite-state portal as activity detection.

Change of State Logic

We gain considerable value by modeling all domain activity as a change of state for two reasons: (1) the memory of what just happened is implicit in the new state, and (2) symmetry, meaning that there is no explicit directionality involved in the change - that is no consideration of cause-effect, or who actually caused the change. Symmetries are always important in software design as we shall see later in more detail.

4.2 Achieving Concurrency in Domains

Software inside a domain is designed in support of some facility, frequently some hardware device, a database, some network protocol, or other subsystem. It is essential that this software be carefully designed to maintain concurrency of the entire system. In the Windows operating system, domains are implemented as Microsoft ActiveX controls, and all domains run in a common thread. Consequently, any procedure call that is termed 'blocking' or 'synchronous' is to be avoided, since it will block all thread activity including all other domains. Associated with every non-blocking call is usually a Callback procedure that can be specified as part of the

domain. This Callback procedure is initiated by the subsystem when it completes some task and the callback routine can then issue more calls to the subsystem or perhaps assertions as described below. It is frequently the case that a facility can best be managed in a separate thread.

4.3 Normalized Interface

To briefly recap the discussion so far, we have cast all lower level software into independent named components called domains, given them a domain independent interface that includes a text string and a binary payload, cast them into a finite-state model (possibly applying fuzzy logic to do so) and we will implement them using conventional programming techniques to guarantee that they all operate concurrently.

There remains the question of how many portal interfaces should there be and what functions should they perform? Before addressing that, let me introduce the notion of a named channel of communication, referred to as a device.

Device Context

Across these portal interfaces, changes of state are ultimately delivered and this begs the question - 'state of what?'. There has to be a representation of the overall state and this representation is referred to as the 'device'. As we shall see, there is a duality with this concept, since 'devices' are also thought of as 'channels' of communication up and down the system hierarchy.

The less that goes into the definition of a device, the better. We go to some length to avoid complex structures or any formal definition of this object, and simply declare that a 'device' is nothing more than a set of property-value pairs, all text strings. There is no order to the list of properties and the only property absolutely required is one called 'name' which will be the objects handle or moniker - the device's name to keep it simple. Two other properties are frequently included: 'class' which identifies the domain that owns the device, and 'type' which identifies a device type within the domain.

The set of properties associated with a device represents its state, and so any change in state is represented as simply 'PropertyName NewState'. Later, we will see how NewState represents a predicate which will form the basis of rule based logic.

Whenever a property value changes, a message is sent from the domain to maintain the device properties across the system, specifically to networked computers described in a later section. Certain properties, however, are special, and changes in

their state are called an assertion, and this leads us to the first generic portal, the Assert function.

Assert

First we apply the symmetry we previously noted above and recognize that assertions have no implied directionality and so we actually require two portal interfaces, one from higher level software into the domain, and the other from the domain up to higher level software. So, assertions can originate from within the domain (e.g., the hardware) or from outside the domain (e.g., a mouse click), both viewed symmetrically. Many software designs destroy this symmetry by using a 'command/notify' semantic, but we shall realize considerable advantage in retaining it later. For now, we simply define the Assert function to be a bi-directional procedure whose string argument is of the form 'deviceName assertion', where the 'assertion' string is interpreted and applied to the devices set of properties.

Domain Independence

The assertion text does not follow any convention. It is any string that conveys the appropriate change of state as intended by the companion component mentioned above. The actual assertion string could be in French or Swahili (or whatever) so long as the domain and its companion agree as to its meaning. We intend that there be no other software in the entire system that will attempt to interpret this string and following this important objective will result in pure domain independence and an associated ability to scale to very large and widely ranged applications.

Domain independence is both powerful and compelling, but it has a small problem. Higher level software may, on occasion, desire to know the state of a property of a device within the domain. For example, a checkbox on a user dialog window needs to reflect the current state of a property value - is it true or false (checked or unchecked)? To do this, we need to interrogate the domain and rather than doing it at higher levels, we let the domain itself do it all, and that brings us to the Evaluate portal function.

Evaluate

The Evaluate portal function couldn't be simpler. String arguments identify the name of the device and an expression of a state condition, and the Evaluate function simply returns true or false depending on whether the device is in that state or not.

Like the assertion string, the string to be evaluated here conforms to no particular format and is formed and interpreted exclusively by agreement between the domain and its companion component. A few other interface functions are required for control and administration that both controls the domain, and creates and destroys devices owned by it.

Implementation

Domains under Windows are implemented as ActiveX components and are dynamically loaded and linked with higher level software as is needed. Windows provides good support for asynchronous I/O to most hardware devices, and most vendors of hardware provide asynchronous runtime support for their products. Asynchronous runtime support generally means non-blocking calls to the device and Callback routines when the device requires service. Concurrency is maintained because the only times the domain is active is (1) in response to a hardware Callback, (2) in response to an assertion from higher levels, and (3) possible Timer functions. Activity from the hardware in the form of Callbacks frequently result in assertions from the domain to higher levels which will possibly cause interaction with other domains discussed in the next chapter.

We have described the lower level of our concurrent system as a potentially large collection of domains - named encapsulations of software in support of specific facilities and subsystems. We have described the requirements for domain independence and its advantages in terms of scalability and applicability. Finally, by standardizing the interface to domains as simple portals that carry strings and binary data, we are assured that we can add domains having wide ranging capabilities without compromising the system's internal design.

4.4 Some Generic domains

If we were to read ahead and discover some of the functions of higher levels in our software hierarchy, we would encounter the need for variables. By variables, I mean those elements of computer languages that hold data like integers, Booleans, strings and the like. To facilitate these requirements and others, there are a few generic domains that are useful across all applications, and we will describe two of them here, namely the String domain and the Timer domain.

String domain

The String domain is a general purpose collection of devices used for data and control. This data is held as a special device property called 'state' and so assertions of the form 'devicename value' basically set the 'state' property to 'value'. Some devices are free text and can hold information like 'The cow jumped over'. Still others hold text data but can only acquire fixed discrete states, like 'one,two,three'. Let's briefly describe each type in detail.

Free Text

This string type is self described in that it holds arbitrary text. assertions to the domain can set the text string to any value, and the Evaluate function tests its current value against a test value.

Discrete Text

This string type can only acquire specific values, such as 'hot,warm,cool,cold' or 'one,two,three'. A particular form of these are Boolean because their discrete choices are 'true,false'. assertions to the domain can set these values and Evaluate can test them.

The properties of a device named Discrete1 having properties value1, value2 and value3, can be modified through the following dialog in Figure 4.1.

Multistate Text

This is a slightly more complex form of the Discrete type. These devices can have a set of parameters, each of which can have selection of states that can be individually set and tested. For example, it might have 'A=1,2,3' and 'B=a,b,c', and 'C=one,two,three' so that assertions can be (say) 'A=2' or 'B=c' or 'C=one', and so on. Some examples later on will make these clearer.

The properties of a multi state device named Multistate1 having states property1, property2, property3, each of which can take the values shown in the Figure 4.2.

Timer domain

In the Timer domain, each device works like an egg timer except that as it counts down, it can assert time values along the way. As an example, a Timer device can have states of 'Start,Ready,Set,Go' and associated with each state is a number of

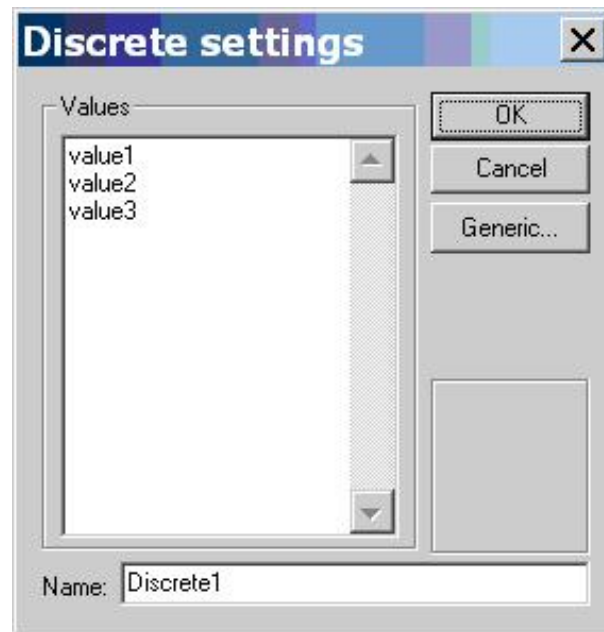


Figure 4.1: Properties of a Discrete device

seconds in time, like '30,10,5,0'. Now, if something asserts 'Start' to the device, its clock will begin at 30 seconds and count down. When the timer reaches 10 seconds, the device asserts 'Ready', and at 5 seconds it asserts 'Set', and finally at 0 seconds it asserts 'Go'.

These generic domains are treated the same as all others even though they are implemented purely in software. In the next chapter we will deal with how to integrate and coordinate the activities of all of these domains into a meaningful application.

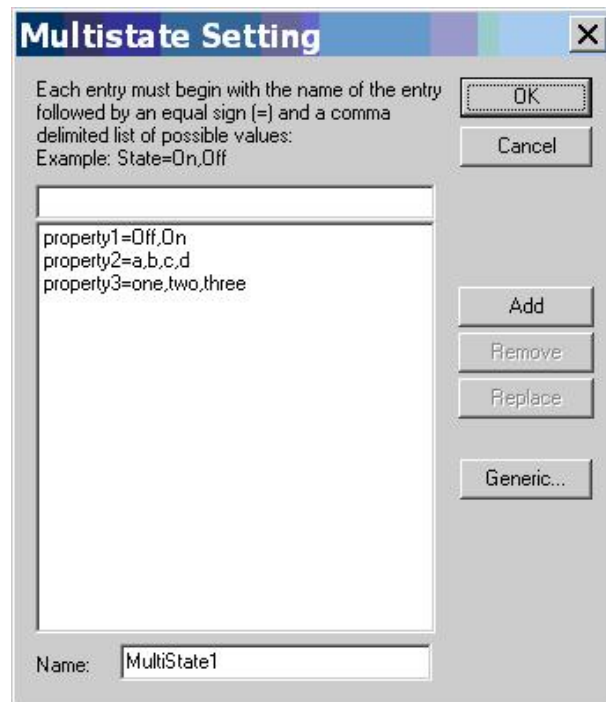


Figure 4.2: Properties of a Multistate device

Chapter 5

Rule Engine: Integrating domain Activity

We now step a level higher in our software hierarchy and find ourselves placed on top of a collection of active domains, each concealing much of their internal processing, revealing to us only changes in their internal state. Devices, which are objects having lists of properties in the form of name-value pairs are used to communicate with the domains. We can change the state of any of the devices by calling the assert function associated with its domain, and we can test for the device's state by calling the domain's Evaluate function.

And while this is going on, any of the domains can assert up to us a change of state presumably initiated by the facility it is managing. How do we deal with all of this activity?

Computers like to do one thing at a time and in a specific sequence, referred to as a procedure. Computer languages such as C, C++, and Basic are called procedural languages. Many applications would tie all of the domains together through the use of a procedural scripting language to be written by an application developer. It is not hard to envision the complexity of this task, and attempts to simplify it, like exhaustive scanning of all inputs, for example, ultimately destroy concurrency. But before we expose the shortcomings of using procedural languages to coordinate the concurrent activity of all of the domains, let's discuss what we may want the overall system to do.

Finite-state systems are frequently described using finite-state diagrams, a graphical depiction of the possible states with arrows showing the various transitions from one state to another. These arrows correlate to the assertions that go to and from our domains. Circles in the diagram represent actual states and correlate to arguments

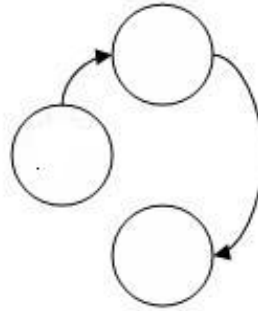


Figure 5.1: State Diagram showing Transitions

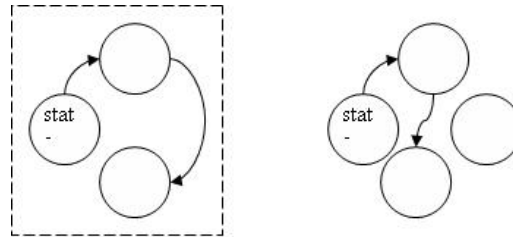


Figure 5.2: State Diagrams of Two Interacting domains

of the Evaluate function of the corresponding domain.

Consider for the moment two domains, each represented by a state diagram as above, placed side-by-side as in Figure 5.2 How do we now describe and implement an interaction between these domains? Conventional methods would suggest using an API (Application Programming Interface) for each and using either a compiled or scripted language, such that an event occurring in one domain would cause some action in another. But relegating the task of 'programming' the interaction of domains has considerable disadvantages. It requires not only intimate knowledge of the domains, but also learning the specifics of the API for each, not to mention learning computer language and how to use it. Moreover, the computer language must be aware of the fact that many such domain interactions are proceeding concurrently and so typically it must scan for all possible events that might occur for all domains.

Instead of employing a programming method, we propose a simpler technique that merely requires domain knowledge. We observe that all interactions must be the result of some state transition, and with a bold stroke of the pen, we simply draw

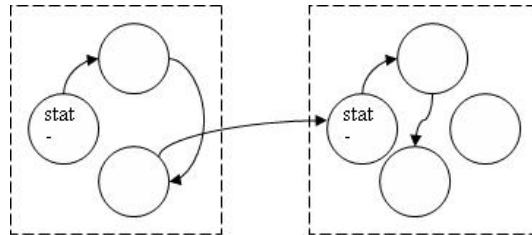


Figure 5.3: Two domains showing a Transition between them

a transition arrow from one domain to the other as in Figure 5.3

Extending this, If we imagine placing all of the state diagrams of all domains side-by-side, we can describe the overall system behavior by 'super transitions' that originate from a state in one domain and point to a state in another. Lets denote this super transition by using the text string from each domain - we need a pair of strings, one representing the state of the originating domain (lets use the term Condition for reasons that will become apparent) and the other representing the transition (we've previously called this the assertion string) in what might be called the target domain. To be clear, we will separate these strings by a vertical bar character:

Originatingdevicename State | Targetdevicename Transition

5.1 Binding

The above expression represents what is referred to as binding between two software domains. It is usually performed by compilers and linkers during development. In certain systems, binding can be performed dynamically, referred to as 'late binding'.

We could describe a very complex system by listing all possible super transitions this way, and the beauty of this list is that it only requires minimal expertise to do so. The construction of the list requires no programming skills, no computer language to learn - simply knowledge of the domains involved, or what is called 'domain -knowledge'.

Another advantage of this list is that there is no order among the elements of the list - each is totally independent. Like a deck of cards, you could shuffle them and the behavior they describe is unchanged. Lacking any order, elements can be modified, replaced, or deleted from the list as atomic operations without affecting the operation of the system as a whole.

The symmetry argument discussed earlier with respect to who causes state tran-

sitions is worth noting again, since the list contains no indication of the past, like how did it get here. However the Condition state was entered, the 'rule' is the same. assert the transition to the target domain. And so, we have encoded the operation and interaction among all of our domains as a list of rules.

It is not unusual that the interaction between domains is itself a complex procedure that may take considerable time. As an example, one domain may make a request to a database domain to lists all database entries satisfying some query, and since this could be a long list, we encode the transfer of each element in terms of one or more rules. In this case, the result of one rule satisfies the condition of another rule, and the system proceeds through a long sequence of rules until the list is completed. We call this a rule-based procedure and we will discuss how to build these procedures later on. It is important to note here, however, that since these procedures are comprised of individual rules, a large collection of such procedures can all operate concurrently. Furthermore, we could take the list of rules that describe a large collection of rule-based procedures, shuffle them like the tunes in your iPod, and the entire system would operate unchanged with a high degree of concurrency.

5.2 Completing the Rule Grammar

If we simply used the rules as we've defined them, we would discover the requirement of creating a lot of extra state variables (String domain) to encode some logic that appears frequently in applications, namely the ANDing of two or more conditions on the left side of the rule. To avoid this, we make our rule slightly more complicated by introducing the AND condition in the rule itself, so that in general, a rule can be of the form

Dev1 Cond1 & Dev2 Cond2 & Dev3 Cond3 | device assertion

Note the '&' symbol used to denote the ANDing of the condition components. This rule will 'fire' only when all of the conditions are true.

Similarly, it is a common occurrence that a rule wants to perform multiple assertions to different domains, and rather than repeating the rule for each, we simply allow the right side of the rule to contain a list of assertions.

Dev1 Cond1 and Dev2 Cond2 | DevA assertion1 and DevB assertion2

Finally, we include a list of assertions to be performed when the left side of the rule becomes false - that is when ANY of the components on the left become false,

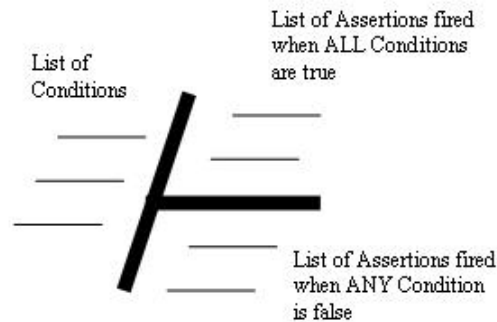


Figure 5.4: Diagram of a Rule showing true and false assertion lists

and we denote this list by the ' ' symbol, and so the complete rule becomes

Condition list | when true list when false list

We diagrammatically show this rule as in Figure 5.4 where the left side is the list of conditions, the upper right quadrant is the list of 'when true' assertions, and the lower right quadrant is the 'when false' assertions.

5.3 Forward Chaining

Rules can be chained together. An assertion appearing on the right side of a rule (true or false list) can be represented as a condition on the left side of another rule, and these connections can flow through an collection of rules, a process known as 'forward chaining'. There are complications in the implementation of forward chaining, such as freezing the systems state until the chain is exhausted. But these are all managed in the design of the procedures that process the rules.

5.4 Rule-based Procedures

Rule based procedures are similar to forward chaining but the chaining actually goes through one ore more domains rather than simply being chained by the rules themselves. In a rule procedure, a domain is handed an assertion which causes it to go do something. This may be time consuming and so the forward chain is temporarily broken. It may be a request to a database or a network transaction, or whatever.

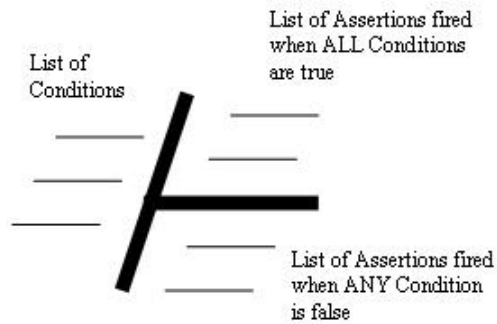


Figure 5.5: Three Rules showing Forward Chaining

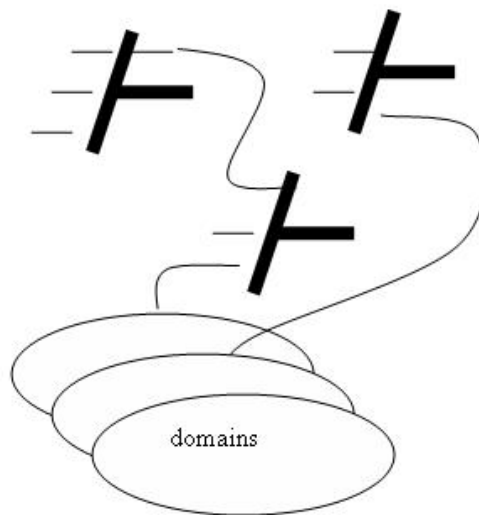


Figure 5.6: Rule Chain Connected to domains

In any case, when the domain accomplishes the request, it performs an assertion to provide notice to the rule engine that the requested task is completed.

As anticipated, there is another rule waiting for this notification which picks up where the previous rules left off and continues the chain. This, then, becomes the procedure, and it can include iterations, forks, merges, and become quite complex. As stated before, any number of such procedures can be operating concurrently - an important attribute since many can take a long time to finish. As previously described, rule components (assertions) can originate from domains as well as be

targeted to domains, and so we show a rule chain with domains in Figure 5.6.

5.5 Comparison to Threads

An example will show the power behind rule based procedures. Suppose an application initiates a procedure that requires three sub tasks – task1, task2, and task3 – and suppose each of them involve some form of delay, like a database request, or a network request. Now, even in a threaded application, it is likely that each of the tasks would be performed sequentially, since the complexity of threads begetting threads is complexity one would rather avoid. But in our rule based system, we simply make three assertions, one for each task, and then wait for a rule which is the AND the assertions each task applies when completed. Each task proceeds concurrently and the user sees reduced latency in the overall operation.

5.6 Bounded Buffer Problem

Throughout the literature on concurrency techniques a recurring example of the problem appears called the 'bounded buffer problem'. This problem is generic to synchronization since it has both a producer of data and a consumer of data who are asynchronously appending and removing data from a common buffer pool of finite size. Blocking should obviously occur if either the producer exceeds the buffer pool size or if the consumer attempts to retrieve from an empty buffer pool. Further, either of them can manipulate the parameters of the buffer, but only to the exclusion of the other so that the logic of the buffer maintains consistency.

Implementing the bounded buffer problem in a message passing scheme is straight forward. Imagine a domain comprised of a buffer pool having assertions append/remove to the domain and accept/data from the domain. To transfer data, the producer would issue an append and the consumer would issue a remove. The domain would assert accept in response to append only if (and later, when) there's sufficient storage available to store the data, and similarly would assert data in response to remove only if (and later, when) data was available in the buffer.

Note that there is no chance that the consumer will interfere with the producer because the domain itself runs in a single thread, even if the messages originated from separate processes or separate computers. It would seem that by focusing on 'data' rather than 'processes' we alleviate certain problems involved in synchronization.

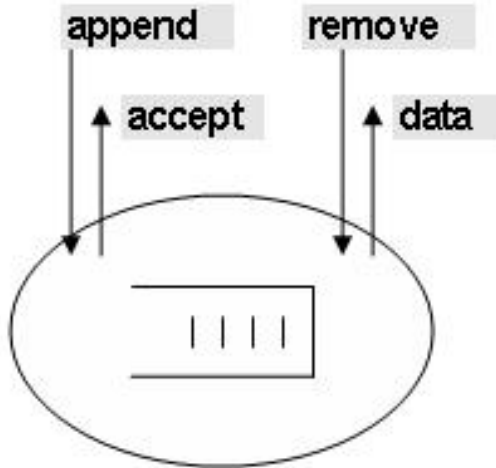


Figure 5.7: Bounded Buffer Problem using Messages

5.7 Discussion of Rule based Languages

Most computer languages, like Basic, FORTRAN and C, are procedural languages meaning that they encode precise time ordered procedures for the computer to perform. Other languages are descriptive languages that encode a description of what to do without regard to how to do it. A subset of descriptive languages is rule-based languages that encode a set of logical rules to describe the systems behavior.

In general, procedural languages, even simple scripting languages, require knowledge of the computer implementation, and frequently burden the user with complex issues of timing, synchronization, keeping track of multiple parallel processes, and so on. Rule based languages, however, bury all of that complexity into the language run time system, and provide the user with a far more elegant framework in which to encode certain types of applications.

The most popular rule based language is Prolog, which has been used in many artificial intelligence applications. There are many other rule-based languages that are integrated into applications, like Mail routing systems for example.

The rule system that we've presented is termed 'event driven' as opposed to 'goal oriented' as in the case with Prolog. In Prolog, if you want the system to perform a specific task, say X, then in effect you establish the goal of preventing everything else in the world to occur except X. This convoluted way of expressing tasks is fine for certain applications, though certainly not ours.

Rules are invariably associated with Boolean logic and many rule processors permit complex expressions of Boolean logic in combinatory form. I have found this complexity to be rather error prone and it not only introduces complexity in the implementation, but worse, introduces logical errors in the system operation. Consequently, we have limited our rule grammar to the simple form presented, having a few side effects that are in fact desirable. Occasionally, the designer of the rules must create intermediate Boolean variables (String domain) in order to hold intermediate state not otherwise derived directly from the domains. Most often, the requirement of creating these intermediate states shed light and provides insight into the behavior of the domain interactions, thereby actually increasing the reliability and correctness of the overall application.

5.8 Consistency

Consistency among rules means that, any given time, all rules are in effect. Maintaining strict consistency among rules is a difficult problem and one that we freely ignore. Let me provide a real world example. Suppose we have a 'security' domain and a 'lighting' domain and we create a rule that says 'when the door opens, turn on the light'. Suppose further, that whenever the light is turned on, a 5 minute timer is started that will cause the light to turn off after it has expired. Now, if the door is still open when the timer expires then we have a dilemma - should we leave the light on satisfying the first rule, or should we turn the light off satisfying the second. To resolve this, we apply rules ONLY on state transitions - called a 'non-persistent' rule application, meaning the rule does not persist beyond the transition that originally fires it. From a practical point of view, this not only greatly simplifies the implementation, but also intuitively gives the desired result in most cases.

5.9 Implementation

The fact that we have cast the problem of interaction among a collection of domains into a set of rules does not remove the fact that we have a computer supported by a language that can only do things sequentially. It does, however, remove the problem from the user or application developer and make the problem subject to some very fast and efficient techniques to arrive at a high performance solution. Specifically, the set of rules are compiled into a binary representation that the processor can run, and this program is called the 'rule-engine'.

In effect, a rule engine, buries all of the if-then-else logic into a fast execution module. The efficiency of processing a compiled binary representation of rules reduces the computation required well below any alternative like scripting languages. The implementation follows the rules of 'never ask a question twice', and do work only when some state changes.

More Symmetry - the 'Server' device

A special device, called the 'Server' embodies the assertions and Conditions of the rule engine itself and provides for a variety of useful system primitives. The Server device looks like any other, although it is implemented quite differently. These are somewhat analogous to internal functions in conventional languages, and include conditions such as the following:

- Timer functions for time-of-day, day-of-week, short intervals, etc.
- Status reporting
- Invoke/Expire rulesets (see below)
- Executing applications

Rule Sets, Compilation, and Dynamic Linking

To better organize a large number of rules, they are organized into named rule sets. Rules within a rule set are compiled as a group, and the system is fast enough to be able to perform a compilation, unlink any previous binary representations of the rule set, and link in the new one. This dynamic linking capability means that rules can be modified and put into service without disturbing the operation of the system.

Rule sets are in their text form are uploaded to client based rule editors which author and modify the various rule components (Conditions and Assertions). For every domain on the server, the rule editor runs its companion component discussed earlier in Chapter 4, and it is these companion components that present dialogs to form the strings representing the corresponding domain's rule component. So, the rule editor itself need know nothing about how to formulate rule elements, and this implements the type checking we discussed earlier.

Dynamic Invocation

Since the rule engine is capable of rapidly compiling and replacing rule sets, it became compelling to introduce the notion of dynamic invocation of rule sets, meaning that under rule control, entirely new rules sets can be loaded and conversely, they can be expired. So, rule sets are no longer static entities loaded at startup, but rather, the system can load rule sets on-demand as the system requires, all under control of other rules designed to do so.

Device Namespace and Server Applications

To facilitate modularity in the application of rules, it becomes convenient to wrap combinations of rule sets and associated devices together in a named object which is called a Server Application. This way, to deploy a capability on a server, the user can install a single named entity, and all of the devices and associated rule sets are installed together. In order to avoid conflicts in device names, a convention has been adopted similar to the namespace defined by the C++ language. device names are prefixed by a namespace and take the form `Namespace::devicename`, using the `::` characters as a delimiter. This namespace is typically the name of the Server Application and eliminates naming conflicts.

We have reached the second level in our software hierarchy, and to briefly recap, our system below us is comprised of a large collection of domains whose activity is coordinated by the operation of a rule engine. The efficiency of the rule engine, together with the ability to perform many complex rule based procedures provides a high degree of concurrent operation. Rules are organized into rule sets and further combined with devices to form named objects that reflect an entire capability or application. We have the ability to load rule sets dynamically, replace them while the system is running, edit them by client based rule editors and provide end-to-end type checking of all Assertions and Conditions for any domain. The entire system is domain independent giving us the ability to incrementally add new domains to the system without interruption.

Chapter 6

Distributed Computing

6.1 Network Architecture

First of all, to state the obvious, networking is important because the facilities we want to manage are geographically dispersed, and the user may not be where we want them to be. The question becomes, how should we develop our network so that the user sees the entire system operate as a coherent entity?

I recall the early networking architecture of IBM introduced around 1970 (SNA) that was designed to be asymmetric, meaning that the link between two computers had a master/slave relationship. Better designs from DEC (DECNET) and Xerox (XNS) showed the advantages of making the link fully symmetric. Symmetry is difficult to achieve; the master/slave approach always had one side in control and was simpler to implement.

These days, asymmetric protocols still prevail on the Internet, but not at the lower computer-computer level, but rather at the client-server level. The popular Browser protocols (HTTP) have a master/slave relationship, putting the Browser in control. Under these constraints, activity is Browser initiated, and if the server has something to say, it must wait until the Browser asks. Now, there are admittedly a host of techniques employed to get around these restrictions, and it is amazing how much capability the industry has pushed through the Browser. But, for these reasons and more, the limitations of the Browser protocols, by themselves, make them unsuitable for a highly concurrent user environment.

The underlying reasons for the asymmetry in HTTP have to do with scalability of the server. In order for the server to scale to thousand of clients and more, it must remain stateless, meaning that each transaction is self contained, having no memory of a previous transaction, and regardless of how many clients are 'connected', the

server retains no knowledge of them. If the client provides some information that might be required on some future transaction, the server says, in effect, "I can't remember all this. Here's a cookie - keep it handy in case I need it sometime". Stateless servers are great for holding information for browsers, but not appropriate for applications of the sort we are describing.

A popular technique of getting around the asymmetry of the Browser is to have it load a plug-in that establishes a back door communication channel with the server, and this alleviates many of the problems. We will discuss the tradeoffs of so-called 'thin clients' versus thick clients later, but for now, we need a fully symmetric protocol, and the popular choice is the Socket protocol of TCP/IP.

6.2 Client-Server Peer-Peer Symmetry

Sockets provide a symmetric, full duplex, reliable data pipe between two computers and we shall build on top of this 'plumbing' yet another set of symmetric relationships. Some computer-computer links connect one server with another or so-called peer-peer connection. Other links connect one or more servers to a client application, or so-called 'client-server' connections. And, at the risk of over using the term 'symmetry', we should not rule out client-client connections.

At the base level of communication between computers, we can envision transactions, or messages going in both directions. At a slightly higher level, we see that some transactions perform functions while others establish context for future transactions. In many cases, the context of a communication link will determine what functions are exchanged. So, it may be useful to formally distinguish context from functions and to derive some elegant methods of managing this context. Let me explain why this is important.

6.3 Managing Network Complexity

We have cast all of the state of our domains into the state objects we've called devices. So, the total state of our system, as we've defined it, is the total state embodied in all of the devices across all of the domains. Consequently, if two of these computers are interconnected, the worst case context would be the sum of the state held in all devices on both machines, attained through a mutual exchange of all the devices residing on both computers.

We note that if you interconnect N computers in a peer-peer relationship, you have established $N(N-1)/2$ connections, and so if each connection adds the context

of its local devices, overall context on each computer grows as N squared. This geometric growth must be avoided if we desire scalability to potentially large numbers of computers.

There is an analogy to computer languages that suggests an interesting solution to this geometric growth problem which I have used. In early FORTRAN applications, one frequently finds large blocks of what is called COMMON data which is shared across the various subroutines. In many applications, these COMMON statements would grow very large, and names of variable had to be carefully chosen to avoid duplication, and each module of the program had to have the identical COMMON statement, and so on. Later, block structured languages like Pascal introduced variable 'scoping' whereby variables existed only within their scope, delimited by Begin/End statements, or in the case of the C language, by '{' and '}' characters. Blocks, and consequently variable scoping, could be hierarchically arranged (blocks contained within blocks) and thereby giving control over the scoping of variables to permit the program's complexity to grow linearly with size. There is a natural tendency to limit the scope of variables to what could be called a 'need to know' basis, and we shall employ this method to network context to allow our systems to scale to very large sizes.

Name Scopes

Instead of using Begin/End or '{}' delimiters to define a scope, suppose we used arbitrary string values, and instead of requiring that scopes be strictly hierarchical, suppose we permitted them to be arbitrarily overlapping. Where we are headed, here, is to assign each device a scope name, and propagate the devices context only to computers which are members of this scope name.

So, every device exists within a named scope. Similarly, each computer has a list of scopes - sort of a membership list. This is a list of all scopes that the computer 'needs to know' about.

There are many great analogies to the use of scope names that manage complexity. For example, the set 'Boston /Middlesex /Massachusetts /USA' is a hierarchical geographic list of scopes. Laws are legislated with this scope so that a resident of Cambridge need not be concerned with a law scoped to Boston, but state and federal laws do apply. I need not bore the reader with further examples of such a compelling concept.

Connecting large numbers of computers with properly managed scoping permits systems to grow to arbitrary size with even sub-linear growth in complexity. In Figure 6.1, the ovals encircle those computers that possess a corresponding named

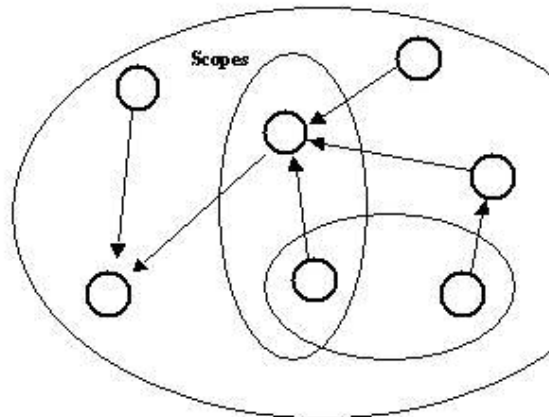


Figure 6.1: Network of Computers Showing Scopes

scope. Any particular computer has a list of scopes which it is a member of. When two computers establish a connection, each computer transmits all devices whose scope name is included on the other computer's list. This information is forwarded through all other previously connected computers as well. So, in effect, any particular device is copied to all computers that lie within the boundaries of the device's scope name.

For this to work, scope regions on the above graph must be simply connected. If there exists two or more separated (multiply connected) regions having the same name, they are treated as independent scopes.

Proxy Devices

Device context that is created on a computer due to a connection with another is termed a 'proxy device'. It is intended that any operation that can be performed on any device can be done on a proxy device as well. So, the 'locality' of a device is distributed across the region of its scope name.

The implementation of the behavior of proxy devices is straight forward. We simply forward all state changes (assertions) across the network, limited by the device scope. There are two special Name Scopes that are reserved for the system: 'Local' and 'Global'. If a device has a 'Local' scope, it will not be proxied across a connection even if the remote client/peer has 'Local' in its list. Also, if a device has 'Global' scope, it will always be proxied across a client/peer connection.

Distributed Rules

Having device context distributed and synchronized across a scoped region of a network means that rules operating on the device can also reside anywhere within the scope region. Obviously, care must be taken to prevent rules in one location conflicting with rules in another, but this concern is small compared to the considerable advantages gained.

As an example, suppose we define a scope to be a particular building, and within that building we have deployed many computers to (say) monitor the various doors throughout the building. Sensors on the doors are correlated to devices, and we desire to apply sets of rules to monitor or control access to the various doors. By assigning all door sensors to the scope of the building, we can deploy the rule set without regard to which doors are connected to which computers.

6.4 Computer to Computer Communications

Reviewing briefly, we have established our design by applying various forms of symmetry, namely:

- No distinction between client-server, server-server, etc.
- no distinction for the computer who initiates the connection (no master vs. slave)
- no distinction for local devices vs proxy devices

Of the many ways to facilitate communication between two computers, we choose the simplest. We pipeline the socket connection with simple messages having the form **MessageType/ StringData /BinaryData**

The `MessageType` parameter is an index of all message types and is used as an argument if a `switch()` statement on each side of the interface. `switch(type) case` Some examples of message types are:

- device context (carried in the string data)
- property updates of a device
- assertions to a device .

On top of this basic message structure, we define a concept called a message thread.

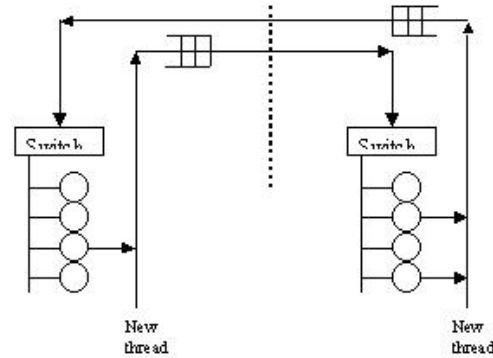


Figure 6.2: Message Thread between two Computers

Message Threads

We define a 'message thread' (not to be confused with C language threads) to be a sequence of back and forth messages over the socket - like a volley in a tennis match. Upon receiving a message on one side of the interface, the computer performs a small amount of processing and then issues another message back. That message, in turn, causes yet another set of exchanges, and this proceeds until some condition is met to cause it to terminate. We formalize this exchange as a 'thread' by creating the message type definitions as a group.

Message Threads can be thought of as small programs or procedures that co-exist across a network boundary. Threads can create new threads based on conditions and thereby create a cascade of parallel threads. A Message Thread is developed by assigning a sequence of Message Type indexes and creating code for them that controls the thread.

The MessageType indexes are carefully designed to accommodate the beginning, looping, testing, and completion of the message thread. As an example, suppose we wish to receive a list of items from a network partner. We create the following sequence of message types, where we've assigned a unique integer (starting at 101) to each:

```

101 send first item
102 send next item
103 here is next item
104 here is last

```

This message thread is initiated by one side issuing type 101. In response to this an arbitrarily long sequence of 102/103 occurs, finally terminated by 104.

Message Threads are either created for specific system functions, like exchanging device context, or for application functions in which case the thread is connected to the rule engine and is therefore programmed by creating appropriate rules as described below.

It should be emphasized that messages do not require a return acknowledgment as this would introduce undesirable round trip delays. The messages are delivered on what is assumed to be an error free link. The socket protocol is implemented on top of lower level protocols that manage network errors. A socket cannot tolerate errors; if it encounters one, it must be destroyed and recreated. This use of a pipelined error free link is a distinct advantage over RPC methods which generally rely on message acknowledgments.

Concurrency among Message Threads

Any number of message threads can be proceeding concurrently. The software that comprises the elements of the thread must be short and fast so as not to delay the processing of messages. For example, a request to query a database should launch the request and immediately return (asynchronous query). Later, when the query returns values (via a Callback for example), the message sequence continues.

So, even very slow procedures can be concurrently performed - procedures to access Internet Web sites, for example. All of them operate concurrently as long as they obey these few simple rules. How are these threads initiated and how are the messages bound to the rest of the system?

Binding of Messages and Message Threads

Certain message threads are initiated by the system. For example, the thread described above for exchanging device context is automatically performed whenever a network connection is established. Actually, this thread is instantiated from both sides of the connection and upon connection; they both proceed independently and in parallel.

Other threads are initiated by the system and never terminate. For example, the initiator of a connection creates a 'ping' thread which simply exchanges ping messages every 30 seconds or so to ensure the link is up and running, causing automatic reconnect on the detection of a failed connection.

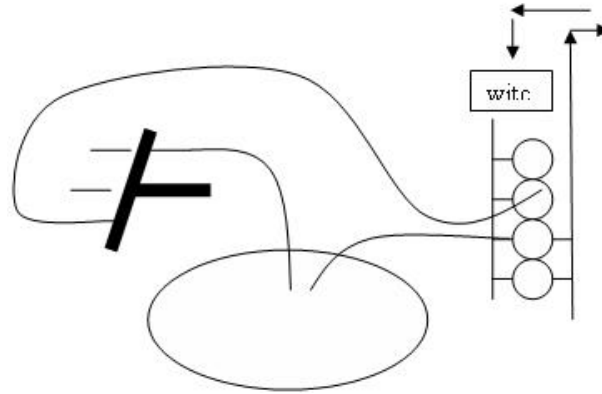


Figure 6.3: Rule Binding Messages and domains

Assertions and property changes originating from domains are bound to the messaging system, as are assertions from the rule engine, so all devices within a scope maintain current state from all domains and rules that operated on scoped devices.

Users can create Message Threads by properly designed rules. Rules comprise the binding between the set of domains on one computer to the set on another computer as shown in Figure 6.3.

Idempotent Messages

Idempotent messages are those that can be redundantly repeated without harm or side effects. For example, delivering the current time upon request is idempotent, even though the redundant time value may be slightly different. However, a message that deposits 100 dollars into your account is not idempotent.

One of the beauties of finite-state systems is that change of state messages are usually idempotent. So, for example, network connections can fail and later be re-established with minimal consequences, since the reconnection establishes the then-current state of all scoped devices.

While idempotency is defined as tolerance to duplicate messages, it is often the case that tolerance to missing messages applies as well. When a message is missed, due for example to a network fault, idempotency is important because, on recovery, the system need not be concerned with whether the message was previously sent - it simply retransmits any message in doubt.

While network errors are one way of missing messages, another more deliberate way is to sub sample them. Here, the system chooses to miss a message, for one

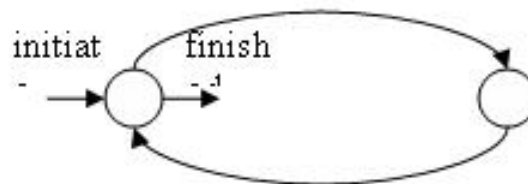


Figure 6.4: Diagram of a Message Thread

reason or another, under the premise that it will eventually send it when conditions permit.

Idempotency can facilitate dynamic network bandwidth management, since, like network failures, if the network becomes congested, we can suspend or slow down messaging as long as we ensure that eventually the system state will be restored. We term this concept 'adaptive network management' and discuss it further below.

Creating Message Threads

Creating message threads should be considered to be a software engineering discipline. Initiating a simple message thread can invoke a time consuming network procedure. Further, since message threads can invoke other threads, there is opportunity to create complete applications implemented entirely by message threads.

The rule engine serves as the control mechanism for threads handling their initiation, testing for conditions, termination, and even error handling. Message threads can relate to other message threads in various ways - one can spawn another, one can be embedded within another, one or more can co-operate with forms of synchronization, and more. There is an apparent programming discipline here that is deserving of a formal representation. A traditional language representation falls short of expressing the implicit concurrency in message threads and so we choose a more graphic representation.

A message thread is shown diagrammatically as a closed circuit with processing at both ends, as in Figure 6.4. The circles usually represent logic carried out by the rule engine. To get the thread going, a rule is fired to initiate the thread, and another rule is fired when finished. Threads can initiate other threads in a cascading arrangement as in Figure 6.5. Two or more message threads can be coordinated by rules as in Figure 6.6. These forms are best explained by way of example and so we

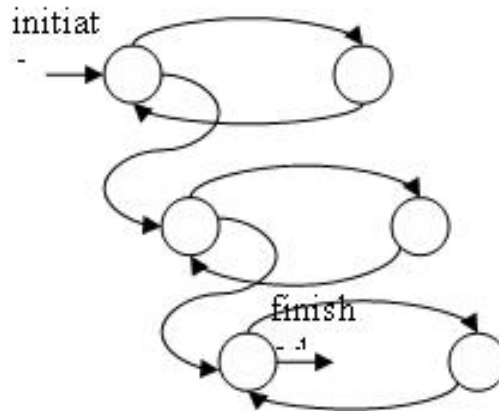


Figure 6.5: Cascaded Message Threads

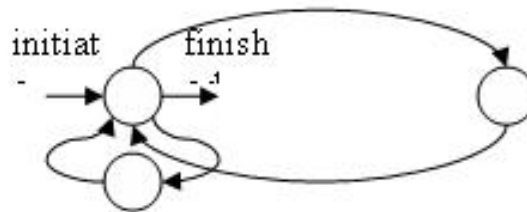


Figure 6.6: Two Coordinated Message Threads

present several application scenarios which typify their usage.

Network Connection, Authentication, Scoping, and Device Context

A good example of cascaded message threads occurs whenever two computers interconnect. Although the connection is functionally symmetric, one side always initiates the connection by creating a standard Socket connection over which the first message thread is begun. This thread performs an exchange of name/password information which is the first level of authentication. Once authentication is established, both sides initiate message threads to communicate Scoping information to the other. Subsequently, both sides initiate a third message thread to convey scoped device

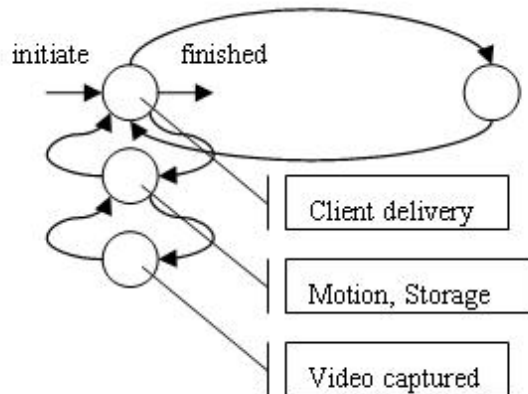


Figure 6.7: Three Stage Coordinated Message Thread

context to the other. Upon completion, the connection is up and running, awaiting further message processing.

Status Monitoring by a Client

An example of coordinated message threads is the monitoring of real-time status information by the client. This example follows the logic depicted in Figure 6.6 where the smaller thread represents high rate real-time collection and logging of data from a domain. The larger, horizontal message thread conveys current values of the data up to a client application. Here, the data rate measured is too high to send synchronously to the client and so the client's message thread sub samples the data collected by the domain's message thread. Since the data is assumed to be idempotent, the effect of sub sampling has minor consequences, and has the additional feature of being adaptive to unanticipated network traffic.

Live Video Monitoring

A further example of coordinated message threads is capture, recording, and viewing of video images from a large number of surveillance cameras.

Here, we show three threads: the first captures and compresses video images; the second compares the image to the previous image to determine if motion has occurred and if so, appends it to disk storage; the third delivers frames to a client, but only under specific conditions, namely that (1) motion has occurred since the

last frame sent, and (2) the client thread has completed the previous cycle - a form of sub sampling as described earlier.

This application places a heavy load on the network, and since the client could be viewing a large number of cameras, coordinating the message threads as described makes the entire system tolerant to varying network loads.

In cases like this where there is a sizable binary payload associated with a message, the message actually conveys references to a common memory storage system rather than actually transferring data. Only when the thread crosses a computer boundary is the data 'marshaled' and instantiated on the other side.

Adaptive Network Management

In these examples and others, we see a common pattern of behavior - idempotent messages are delivered on two conditions: first that there has been a change of state (i.e., meaningful data to send), and second that the network is prepared to deliver it. As the network becomes congested, the network message threads naturally slow down, reducing the demand on the network, thereby automatically adapting to whatever bandwidth is available, including none in the case of a temporary failure. In effect, it automatically sub samples the information without adverse side effects.

There is value in formalizing the association between idempotent messages and adaptive network management. Reducing the non-critical load on a network is one of the best ways to maintain concurrency at the user level. By recognizing that real-time data acquisition, finite state transitions, and more, are all variants of idempotent messages, network based applications can scale to very high levels without encountering critical errors.

6.5 Summary

At the third level of our application hierarchy, we have a network of both peered and client computers whereby state generated from low level domains is consistently maintained across the 'scope' of connected computers, each processing rules that coordinate and integrate this distributed activity. At no point is the system blocked, waiting for resources, and at no point does one activity interfere with another as long as the raw processing capabilities of the computers are sufficient. Further, considerable robustness is maintained throughout the systems and everything, including network bandwidth, is dynamically managed. There are no strong bindings of one software module to another, nothing that need be compiled and linked together, other than the core implementation. This implementation is independent of the do-

mains it manages, and so, these domains can come and go without interruption and in fact new domains can be deployed without interruption. Rules that integrate domain activity can be added or modified without interruption. Scope lists on servers can be dynamically managed to determine the set of computers that coordinate any particular device's activity.

Our fourth level of concurrent processing lay in the client.

Chapter 7

Client Applications

At the lowest level in our hierarchy, we developed the notion of a domain to implement the characteristics of real world facilities. Now, at this higher level we need something equivalent to implement the characteristics of the human user.

The field of Human Computer Interaction (HCI) is an active area of research and beyond the scope of this book. A brief discussion of the issues in HCI, however, is useful in establishing a context for the design discussions that will follow, in part because it will validate the difficulties in prescribing what the use interface should be like. We will need a platform that can accommodate a wide range of human interaction.

Historically, client applications were ordinary applications that happened to rely on one or more servers for information. When the Browser became popular, it became known as a 'thin' client, relegating ordinary clients the term 'thick' client. More recently, the Linux world, has popularized the term 'stateless' client. We need a taxonomy.

7.1 Stateless Clients

In a traditional computer application, the state is located on the client computer. It is installed on the client, it resides on the client, and usually, the data it uses also resides on the client, though it may use servers as well. In contrast to this, a 'stateless' client derives its state from the server.

- Pros: The advantage of stateless clients is that the application can be centrally managed at the server, regardless of how many clients there are.

- Cons: One disadvantage of stateless clients is that the application must conform to the capabilities and limitations of the client program, and this is not always easy to do. Another disadvantage of stateless client applications is the extra work to set up, or 'host', the application on a server.

7.2 Thin Clients versus Stateless Clients

The term 'thin' client is technically synonymous with 'stateless' client, but has been strictly associated with a particular stateless client, the ubiquitous Browser application.

- Pros: The principal advantage of Browser applications is that the Browser is freely distributed on all PCs these days, and so there's nothing to install or maintain on the client. Another advantage is that it is very good at doing what it says - namely browsing for information.
- Cons: asymmetric protocols, minimal client-side context

As anticipated, the principle disadvantage is that the Browser isn't designed to perform all applications well. A few years ago, it was predicted that all applications would be Browser based and yet most of my office tools like word processors and spreadsheets remain thick clients. In fact, following the 'bubble' of Browser applications, one might spot a trend back toward thick clients such as Napster, [examples] etc. Choosing the right Client The choice of which approach best fits an application becomes fairly clear when one examines (1) the complexity of the application, (2) where the application state resides, (3) requirement for centralized (server based) control, and more. In our case, the servers we've described earlier are rich in application context and so some form of a stateless client seems appropriate. Further, there is no doubt that the Browser can implement applications well beyond its original purpose as an HTML viewer. However, for an application tightly bound to real-time acquisition and control, where the concurrency of a lot of independent activity is viewable, the best choice seems to be a thick, stateless client.

At the base of our client we have the bi-directional message loop previously described, capable of participating in message threads, as well as having direct interaction with domains as well as the rule engine. Let's begin a description of the client by describing a simple application example that exploits much of the server implementation we've presented so far.

Example - Distributed Database Query

Suppose we have a number of servers, each running a database domain of (say) class `MyDatabase`, and suppose all of these servers are within the scope called (say) `'MyData'`. When the client creates a single device of class `MyDatabase`, having a scope `MyData`, and say a type of `'Query'`, the servers will propagate the proxies of the device to all servers within the scope. Then, when the client asserts SQL text to the device, this change-of-state will in turn be asserted to all scoped computers causing each to issue the query to their local database. Results of the query asynchronously arising from the various databases will be asserted upward through each server's message loop, making its way up to the client's message loop where the results of all servers are processed.

This demonstrates the simplicity in performing what would otherwise be a complex and usually serialized process, hitting one database at a time. When analyzed in detail, the execution of this distributed query is optimal, with no unnecessary delays, and while it is proceeding, all other functions are proceeding in parallel.

7.3 Components of the User Interface

The request for information from one or more databases is one of numerous components of a user interface. Other examples of components include live video feeds, status indicators, real-time plots and graphs, control panels with buttons and other widgets, voice recognition, animations, and many more. Clearly, we need a flexible approach to this, since each application environment will be customized to their purpose. We introduce the component platform.

Component Platform

A platform is something to build on, and we will describe a platform designed to hold an assembly of graphical components that will comprise a user interface. These components will be assembled from an extensive library, and the selection of components and their arrangement will be determined either by the server or by the client itself.

Each component will appear to be active and independent of others. Their activity will be driven from a message processing loop managed by the platform, permitting GUI components interactivity with both server-based domains as well as other components in the user interface. The platform itself acts as a message switch. It receives messages from all components and routes them to other components. One

such component connects to servers and facilitates message traffic to/from one or more servers.

Server Component

By 'server component' we mean the software component within the client application that provides connection to one or more servers. And this requires a little more explanation, because as we've previously discussed, servers can be peer connected in which case a single connection from the client in effect connects indirectly to all of the peers. But the server component we are discussing here affords a higher level of concurrent connections in that the servers in this case are not peer connected, but rather disjoint. So, from our client, we can have multiple connections to multiple servers, each of which could be peered indirectly to more servers.

In most client/server applications, the servers are designed to concurrently process transactions from any number of clients; however, it is usually the case that a client can only connect to a single server. The reason for this restriction may seem surprising. It is that the servers are indistinguishable.

"Wait!" you say, "I can clearly distinguish ebay.com from cnn.com what's the problem?" The problem is that although you can distinguish two web sites, your Browser cannot. If it connected to both, it could not naturally merge them together, so it creates some artificial method like placing them in separate instances of the Browser, or in separate Tabs of the same instance. Browsers can reference multiple sites from standard HTML by placing their content in separate Frames, but in all cases, the information is factored into independent visual segments that the user must arbitrate.

From the server's perspective, clients are mostly indistinguishable, although for the duration of a transaction, the client's source address is what is used to distinguish them. For the most part, servers have little need to provide integrated services among multiple clients with some notable exceptions. Applications such as Instant Messaging require that one client talk to another, and here we discover the solution to integration, namely that clients of Instant Messaging must have unique names.

Unique Naming

In order to fully integrate the activity of multiple servers, we must uniquely name all of their context, particularly the names of devices they manage. To do this, we follow the naming convention used in the C++ language by defining each server a string that identifies its namespace, and then prefix this string to all device names

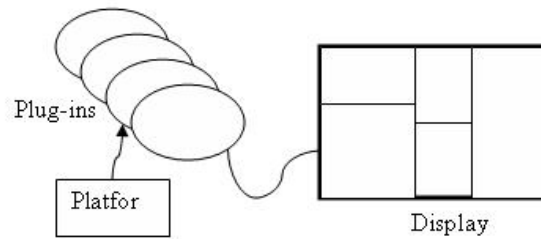


Figure 7.1: Multiple GUI Plug-ins Managed by the Client Platform

on the client by using the convention:

Servername::devicename

If the device is already a member of a namespace within the server, we use a 'dot' notation as follows:

Servername.Subname::devicename

An alternative approach to the '::' naming convention is to encode a server ID into each and every device name itself. In either case, unique naming will guarantee that entities from a collection of servers can freely intermingle without naming conflicts, and this gives our client application some special capabilities.

So, if your application monitors the buildings of a dozen field offices, your client application can connect to all of them world-wide and reference a list of (say) "MainEntrances" without conflicts.

GUI Components

A GUI (Graphical User Interface) component is typically one of many components that the platform places within the window area of the client. The assembly of GUI components forms a mosaic of seamlessly adjoining rectangles and each component renders independent, concurrent activity. The platform's message switch routes messages from the server components to all GUI components and vice versa. Figure 7.1 shows an example of the user's display. Each rectangular area is a plug-in. Live video is a particularly demanding task.



Figure 7.2: Sample GUI Showing Multiple Plug-ins

Chapter 8

Putting it all Together

We have developed at least five levels of software hierarchy in connecting our user to a highly parallel concurrent world. From top to bottom, we've presented:

- A mosaic of graphical plug-ins, each operating concurrently
- A platform of messaging tying plug-ins to multiple servers
- A network of servers, peer connected with shared context
- A rule processor optimized to integrate activity from domains
- domain modules specifically designed to manage various subsystems

Throughout the design of this hierarchy, we have employed a robust message passing technique to maintain system performance, together with a fast rule engine, providing the user with a level of interaction and perception of real-time operation across the entire distributed system.

Index

- Apollo Computer, 12
- Assert, 23, 25, 26, 29, 32, 35, 57
- binding, 31, 48
- Comrie, Leslie, 8
- domain, 19–26, 29–35, 37–39, 42, 48, 51–53, 55–57, 61
- Dynamic Linking, 38
- EDSAC, 8
- EDVAC, 8
- ENIAC, 7–9
- Evaluate, 23, 25, 29, 30
- finite state, 21, 52
- finite-state, 4, 20–22, 29, 48
- fuzzy logic, 21, 22
- Goldstine, Herman, 7
- Message Oriented Middleware, 4, 13
- message thread, 4, 45–47, 49, 50
- name scope, 4, 43, 44
- Network Computing System, 12
- NORC, 11
- proxy device, 44, 45
- Remote Procedure Call, 5, 12
- rule engine, 4, 38, 39, 47, 49, 56, 61
- sockets, 42
- Sun Microsystems, 12
- Turing, 9
- Turing, Alan, 8
- von Neumann, 7–9
- Wilkes, Maurice, 8