Programming, Concurrency and Client-Server Computing

School of Computing, Engineering and Mathematics
University of Brighton

Revision Booklet

May, 2012

Contents

Programming Languages

There are many different paradigms in computer programming here are some examples

Imperative C

Declarative SQL, CSS

Object Orientated Ada, C++, C#, Java

Logical Prolog

Functional Haskell, F#, Lisp, Clojure (if your a real hipster)

A comparison between Java, Ada, Haskell and Prolog

Typing Systems

Static

A programming language is said to use static typing when type checking is performed during compile-time as opposed to run-time. Statically typed languages include ActionScript 3, Ada, C, D, Eiffel, F#, Fortran, Go, Haskell, haXe, JADE, Java, ML, Objective-C, OCaml, Pascal, and Scala. C++ is statically typed, aside from its run-time type information system. The C# type system performs static-like compile-time type checking, but also includes full runtime type checking.

Static typing is a limited form of program verification (see type safety): accordingly, it allows many type errors to be caught early in the development cycle. Static type checkers evaluate only the type information that can be determined at compile time, but are able to verify that the checked conditions hold for all possible executions of the program, which eliminates the need to repeat type checks every time the program is executed. Program execution may also be made more efficient (e.g. faster or taking reduced memory) by omitting runtime type checks and enabling other optimisations.

Dynamic

A programming language is said to be dynamically typed when the majority of its type checking is performed at run-time as opposed to at compile-time. In dynamic typing values have types, but variables do not; that is, a variable can refer to a value of any type. Dynamically typed languages include APL, Erlang, Groovy, JavaScript, Lisp, Lua, MATLAB, PHP, Prolog, Python, Ruby, Smalltalk and Tcl.

Implementations of dynamically-typed languages generally associate run-time objects with "tags" containing their type information. This run-time classification is then used to

implement type checks and dispatch overloaded functions, but can also enable pervasive uses of dynamic dispatch, late binding and similar idioms that would be cumbersome at best in a statically-typed language, requiring the use of variant types or similar features.

More broadly, as explained below, dynamic typing can improve support for dynamic programming language features, such as generating types and functionality based on run-time data. (Nevertheless, dynamically typed languages need not support any or all such features, and some dynamic programming languages are statically typed.) On the other hand, dynamic typing provides fewer a priori guarantees: a dynamically typed language accepts and attempts to execute some programs that would be ruled as invalid by a static type checker, either due to errors in the program or due to static type checking being too conservative.

Dynamic typing may result in runtime type errorsthat is, at runtime, a value may have an unexpected type, and an operation nonsensical for that type is applied. Such errors may occur long after the place where the programming mistake was madethat is, the place where the wrong type of data passed into a place it should not have. This may make the bug difficult to locate.

Dynamically typed language systems' run-time checks can potentially be more sophisticated than those of statically typed languages, as they can use dynamic information as well as any information from the source code. On the other hand, runtime checks only assert that conditions hold in a particular execution of the program, and the checks are repeated for every execution of the program.

Development in dynamically typed languages is often supported by programming practices such as unit testing. Testing is a key practice in professional software development, and is particularly important in dynamically typed languages. In practice, the testing done to ensure correct program operation can detect a much wider range of errors than static type-checking, but full test coverage over all possible executions of a program (including timing, user inputs, etc.), if even possible, would be extremely costly and impractical. Static typing helps by providing strong guarantees of a particular subset of commonly-made errors never occurring

Strong Vs Weak

Most generally, "strong typing" implies that the programming language places severe restrictions on the intermixing that is permitted to occur, preventing the compiling or running of source code which uses data in what is considered to be an invalid way. For instance, an addition operation may not be used with an integer and string values; a procedure which operates upon linked lists may not be used upon numbers. However, the nature and strength of these restrictions is highly variable.

Listing 1: Weak Typing

```
1 a = 2
2 b = "2"
3
4 concatenate(a, b) # Returns "22"
5 add(a, b)         # Returns 4
```

Listing 2: Strong Typing

```
1 a = 2
2 b = "2"
3
4 concatenate(a, b)      # Type Error
5 add(a, b)              # Type Error
6 concatenate(str(a), b) # Returns "22"
7 add(a, int(b))          # Returns 4
```

Java

Java is a static, Object Orientated , structured, imperative, generic, reflective generic reflective language.

The language derives much of its syntax from C and C++ but has a simpler object model and fewer low-level facilities. Java applications are typically compiled to bytecode (class file) that can run on any Java Virtual Machine (JVM) regardless of computer architecture. Java is a general-purpose, concurrent, class-based, object-oriented language that is specifically designed to have as few implementation dependencies as possible. It is intended to let application developers "write once, run anywhere" (WORA), meaning that code that runs on one platform does not need to be recompiled to run on another. Java is currently one of the most popular programming languages in use.

- Multi threading
- Networking Libraries

Ada

C++

Haskell

Haskell is a functional, lazy/non-strict, modular language.

Haskell features lazy evaluation, pattern matching, list comprehension, type classes, and type polymorphism. It is a purely functional language, which means that in general, functions in Haskell do not have side effects. There is a distinct type for representing side effects, orthogonal to the type of functions. A pure function may return a side effect which is subsequently executed, modeling the impure functions of other languages.

Haskell has a strong, static type system based on HindleyMilner type inference. Haskell's principal innovation in this area is to add type classes, which were originally conceived as a principled way to add overloading to the language, but have since found many more uses.

The type which represents side effects is an example of a monad. Monads are a general framework which can model different kinds of computation, including error handling, nondeterminism, parsing, and software transactional memory. Monads are defined as ordinary datatypes, but Haskell provides some syntactic sugar for their use.

- Recursion over iteration, Haskell does not have loops and because of this it uses recursion iterate through lists. Haskell makes use of tail cull recursion which allows the

program to recurse indefinitely with out causing a stack overflow that would occur in traditional imperial and object orientated languages.

- Sparks, Haskell Threads and Worker threads. Haskell deals with concurrency in a unique way. when a task is to be performed it turned into a spark. A Spark is a essentially a task that is added to a task queue. There sparks are then ignited by Haskell's threads which are then actioned by a worked thread. This way of delegating tasks offerers very efficient multi core performance which very minimal over head in code due to the fact the complex delegation of how to perform tasks is calculated automatically.

Prolog

Prolog is a dynamically typed,general purpose logic programming language associated with artificial intelligence and computational linguistics. In Prolog logic is expressed as sets of facts and rules.

Prolog features a built in back tracking engine which allows of easy evaluation of large rule sets, this gives Prolog a form of overloading on rules.

Prolog is tail call recursive with iteration handled in a similar way to Haskell.

Prolog has many different implementations SWI-Prolog and Win-Prolog are two of these. Some Prolog implementations support networking which allow the creation of distributed systems.

Concurrency

What it is

Concurrency allows you to effectively utilise available processing power by allowing a computer to perform several different tasks simultaneously (e.g web web server requests, spell-checking whilst typing)

Types of concurrency

Coroutines With co-operative schedules (or coroutines), processes suspend voluntarily when they need to wait for an event. Processes may also suspend at regular intervals during lengthy processing.

Timeslicing With pre-emptive scheduling (or timeslicing), processes are suspended as a result of interrupts from external hardware. This can present problems with shared resource access, as it is impossible to predict at what point a process will be suspended.

How it's implemented

Java

Threads Java's *Thread* class can be used to implement concurrency. Each instance of the class represents a different 'thread' of execution, allowing multiple processes to execute simultaneously.
Useful methods: *Thread.start()*, *Thread.run()* - thread halts on exit from run, *Thread.sleep(time)*, *Thread.interrupt()*, *Thread.get/setPriority()*

Synchronisation Problems will occur if two threads try to modify/access shared data simultaneously (or rather if one tries to access while one tries to modify, or both try to modify). Using a *synchronized* block prevents data corruption, as only one thread is able to execute it at a time. Each Java object has an internal lock and a queue for waiting threads; if the lock is clear, lock the object and enter the block, if the lock is set, wait. On exit from the block, clear the lock and wake up a waiting thread (if there are any). This isn't very OO, as data is only protected where it is accessed, not where it is defined. The solution is to 'synchronize' an entire method.

Ada

Tasks

Tasks are Ada's equivalent of Java's threads; each separate task representing a separate 'thread' of execution. Tasks can communicate with each other via 'entry' calls. Entry points are declared in the task spec a task declares entries (like procedures) in the task spec in the form of accept statements. An entry call causes a rendezvous between the called task and its caller. The caller waits for the task to accept the call the accept statement waits for a call (see Listing).

Listing 3: Ada's Accept Statement

```
1 loop
2         select
3                 accept Show (Message : in String) do
4                         Put_Line(Message);
5                 end Show; or
6                 accept Stop;
7         exit;
8         end select;
9 end loop;
```

Entries require a client/server relationship, as they only communicate between one task and another. This is inefficient when just accessing shared data (tasks must be rescheduled to actually pass the data).

Protected records

Protected records (added in Ada 95) use shared data much more efficiently. Protected records contain private data which is accessed by public functions, entries and procedures.

Functions are given read-only access to the data and can be executed by multiple tasks simultaneously.

Procedures have read and write access to the data, but can only be executed by a single task at any one time. Additionally, no other tasks can be executing any other functions, procedures or entries while a procedure is executing.

Entries are similar to procedures, but also has a guard condition which suspends the caller until it evaluates to true.

Haskell

Haskell supports both pure parallelism and explicit concurrency, however as rule of thumb use Pure Parallelism if possible, otherwise use Concurrency.

Parallelism

Pure parallelism (Control.Parallel), can be used to speed up pure (non-IO) parts of the program by speeding up a pure computation using multiple processors. Pure parallelism has these advantages:

- Guaranteed deterministic (same result every time)
- no race conditions or deadlocks
- Software Transactional Memory (STM)

The following code has been parallelized in haskell, this has been done do by importing the Control.Parallel libary and by using **par** and **pseq** combinators.

par tells the compiler that the values can be calculated in parallel

pseq forces the evaluation of c

Listing 4: Parallel code in haskell

```
1
2
3  import Control . Parallel
4
5  main = a `par` b `par` c `pseq` print (a + b + c)
6  where a = ack 3 10
7  b = fac 42
8  c = fib 34
9
10 fac 0 = 1
11 fac n = n * fac (n -1)
12
13 ack 0 n = n +1
14 ack m 0 = ack (m -1) 1
15 ack m n = ack (m -1) ( ack m (n -1) )
16
17 fib 0 = 0
18 fib 1 = 1
19 fib n = fib (n -1) + fib (n -2)
```

Map is a high order function that applied a given function to each element in a list, this is often referred to as *apply-to-all* , In haskell this is written as map *function* list.

Reduce also known as fold, is a method of optimizing operations on lists for instance.

The folding of the list [1,2,3,4,5] with the addition operator would result in 15, the sum of the elements of the list [1,2,3,4,5]. To a rough approximation, one can think of this fold as replacing the commas in the list with the + operation, giving 1 + 2 + 3 + 4 + 5.

In the example above, + is an associative operation, so the final result will be the same regardless of parenthesization, although the specific way in which it is calculated will be different. In the general case of non-associative binary functions, the order in which the elements are combined may influence the final result's value. On lists, there are two obvious ways to carry this out: either by combining the first element with the result of recursively combining the rest (called a right fold), or by combining the result of recursively combining all elements but the last one, with the last element (called a left fold). This corresponds to

a binary operator being either right-associative or left-associative, in Haskell's or Prolog's terminology. With a right fold, the sum would be parenthesized as $1 + (2 + (3 + (4 + 5)))$, whereas with a left fold it would be parenthesized as $(((1 + 2) + 3) + 4) + 5$.

In practice, it is convenient and natural to have an initial value which in the case of a right fold is used when one reaches the end of the list, and in the case of a left fold is what is initially combined with the first element of the list. In the example above, the value 0 (the additive identity) would be chosen as an initial value, giving $1 + (2 + (3 + (4 + (5 + 0))))$ for the right fold, and $((((0 + 1) + 2) + 3) + 4) + 5$ for the left fold.

Concurrency

Concurrency (Control.Concurrent): Multiple threads of control that execute "at the same time".

- Threads are in the IO monad
- IO operations from multiple threads are interleaved non-deterministically
- communication between threads must be explicitly programmed
- Threads may execute on multiple processors simultaneously
- Dangers: race conditions and deadlocks

Issues Associated With Concurrency

Although concurrency adds the power to do new things, it also brings with it new types of errors.

Deadlock

Deadlock occurs when two (or more) processes require access to an inaccessible certain resource in order to continue. This usually occurs when one process has a lock on some resource which is needed by another process, which itself has a lock on a resource which the first process holds. Neither process are able to relinquish their lock on the problem resources, as they cannot get a lock on the new resources etc. etc.

All four of the following conditions are necessary for deadlock to occur:

1. Tasks need to use a non-shareable resource
   Can be prevented by virtualising resources (e.g. print spooling on disk: printer is non-shareable, disk is shareable), though this is not always possible (e.g. a railway track is not shareable between two trains and cannot be virtualised).

2. Tasks hold onto resources while waiting for extra ones
   Insist that all resources are allocated at once (task cannot proceed until all resources have been granted). This is inefficient as resources will be allocated when not needed.

3. Resources cannot be taken away from tasks by a third party
   See above solution.

4. There is a circular chain of tasks requesting a resource held by another task
   Resources can be prioritised, allocated in priority order. A process must finish using (and release) high priority resources before it can use a lower priority one.

It is not always possible to recover from deadlock. The operating system may check for deadlocks by checking the thread table for circularities. If a deadlock is detected, the OS will kill one of the locked processes until the deadlock is broken. This can obviously have severe implications for the program. Similarly, if deadlocks are not dealt with within the program, it may become unresponsive, forcing the user to kill the program manually.

Livelock

Livelock is similar to deadlock, except that tasks are still able to proceed. However, execution is useless in that tasks will not be able to make any meaningful progress. For example, ethernet, where collisions cause back-offs of exponentially increasing length (this variation in wait time will normally eventually break the lock). Where contention is low, probability of livelock is low enough to ignore (although this is not a suitable response in safety-critical situations). Livelock isn't as easily definable as deadlock, the system may appear to be functioning. Deadlocked threads cannot be scheduled, whereas livelocked ones can. If a 'fair' scheduling algorithm is used, livelock can be avoided (e.g. if a guarantee is made that every request is eventually dealt with).

Starvation

Starvation occurs if one or more tasks are 'starved' of resources by other tasks. This might result from a poor choice of task priorities so that high priority tasks will hog resources that lower priority tasks also need, meaning that the lower priority tasks are never able to function. Even if the high-priority thread is blocked, the low-priority thread still cant get hold of the resource it needs.

Priority inversion

This occurs if a high-priority task (A) is unable to access a resource which is held by a lower-priority task (B). If A suspends until the resource is free, B is able to proceed. This means that a low-priority process is taking precedence over a high-priority process. If thread A waits in a loop for the resource, the result is a perpetual livelock: B never runs because A is running, but A is always waiting for B.

The Dining Philosophers

> *Five philosophers go to a Chinese restaurant and are seated round a circular table. There is a single chopstick between each pair of philosophers. Each philosopher alternately thinks (which involves putting down any chopsticks that*

the philosopher is holding) and eats (which involves picking up two chopsticks - only one chopstick can be picked up at a time).

The dining philosophers is an example of a problem which exhibits the main dangers associated with concurrent systems. **Deadlock:** Each philosopher picks up left chopstick and waits forever for the right chopstick to reappear. **Livelock:** Same as above, but put down left chopstick if right one is unavailable. **Starvation:** Two philosophers can starve another sitting between them if they are never thinking at the same time

Solving the problem: Deadlock can be avoided by providing one extra chopstick or one fewer chair, having one left-handed philosopher, allowing philosophers to snatch chopsticks from each other, philosophers put chopstick down if the other isnt available, or allowing chopsticks to be shared.

Livelock only requires circular waiting for unshareable resources, so can be avoided by picking up left chopstick and put it down again if chopstick not available, or snatching chopstick from neighbours.

Another solution could be to provide a bottle of soy sauce, whereby philosophers can only pick up chopsticks if they have the bottle, sauce is put down sauce after both chopsticks are collected. The bottle ensures mutual exclusion for the critical act of picking up a chopstick.

Distributed Systems

Multi-processors

Are becoming increasingly common due to their decreasing cost; adding extra processors (or cores) is a much more cost-effective way of increasing performance. Many embedded real-time systems are ad-hoc distributed systems, configuration determined by overall structure of system being controlled (e.g. aircraft: cockpit systems, fuel systems, wings, tailplane). Processors must be located where they are needed and networked together.

Tightly coupled Multiple processors sharing a single memory (e.g. SMP, quad Pentium motherboards). Tightly coupled systems are good for performance (i.e. more processors = faster), but bad for fault tolerance, as the system relies on shared resources. Tightly coupled systems require simpler coordination of tasks, as have shared memory (using some form of semaphore system).

Loosely coupled Distributed networks with no shared memory, with nodes (processors) communicating over a Local Area Network. This makes coordination more difficult (as there's no shared memory), but also means that fault tolerance is possible by introducing duplicate nodes. An issue which is introduced with loosely-coupled systems is that of whether to use homogenous (with identical processors) or heterogenous (different processors) systems. Homogenous processors are able to compare results at regular intervals, using a 'majority' voting process allows any failing nodes to be identified. However, if the voting is done through a central unit, failures in that unit will fail the whole system. This can be solved using a distributed analog voting mechanism whereby each unit generates an analog signal (e.g. a voltage) which is applied to one or more common carriers. The resultant voltage indicates result. Introducing heterogenous systems can introduce extra communication difficulties based on the systems' architecture (e.g. using big endian vs. little endian). Fault tolerance is also more difficult, as processor speeds will likely vary, development costs are much higher, and truly independent development is difficult to achieve.

Example Use-case: The (Now Defunct) Space Shuttle Uses four main processors in a homogenous system, and uses majority voting. If one processor fails, it leaves three others while it is being replaced, the remaining three can continue majority voting. Also has one heterogenous backup processor which is used in emergencies when two of the four main processors have failed (kept as a warm standby, monitoring the main processors to keep up to date).

Problems

Distributed systems introduce a number of issues which must be addressed in order to work successfully. One of these issues is global state; for a distributed system to work effectively, there must be shared memory or global state. There also needs to be some common timebase so that the order in which events occur can be determined. Among other things, this is required in order to 'snapshot' the complete system in its current state.

Another big problem with distributed systems is that they are nondeterministic (i.e. the outcome can be unpredictable due to timing issues). There are two types of nondeterminism: angelic (an angel/external agent ensures that the best choice is always made), and demonic (no assumption can be made about the choice the demon will make). Angelic determinism is generally impossible to achieve, were it possible, any NP-complete problem could be solved in polynomial time by always selecting the correct path to follow. The alternative is to add mechanisms to make things deterministic.

Issues to be addressed:

- Ensuring a consistent timebase so that all processors can tell what order events occur in.
- Ensuring shared resources are not accessed by multiple processors at the same time (distributed mutual exclusion).
- Need to be able to recognise situations where one processor deadlocks another (distributed deadlock detection).

Failure modes: transient, permanent, intermittent; fail safe (known state), fail soft (graceful degradation); fail uncontrolled, fail late (correct values but too late), fail silent (no values), fail stop (no values, failure visible to other nodes) *Uncontrolled intermittent failures are the hardest to deal with, as parts of the system will act 'traitorously'*

Global Time

Due to unavoidable communication delays, it is impossible to determine absolute time on any node. In most cases, it is sufficient to simply have a correct list of the order in which events occur within the system (generally absolute time doesn't matter). This sequence can be used to number events and provide a non-absolute timestamp. As an example, determining if event A happened before event if A and B originate from the same node and timestamp A ¡ timestamp B then yes, also yes if A is a message being sent and B is that message being received, yes if timestamp A ¡ timestamp C and timestamp C ¡ timestamp B. Otherwise, we dont know. Upon receipt of a message, the local timestamp is incremented to be higher than that of the incoming message (this technique only suitable for two nodes).

With networks of more than two nodes, 'vector clocks' can be used. With vector clocks, each node maintains a vector of timestamps, one for each node. Along with each message, a complete vector is also sent. The receiver of the message updates their own vector so that each timestamp is greater than the one in the message

Network issues

With regards to network issues, we must assume that the communications network is reliable (i.e. any messages sent will eventually be received). This can be done by requiring acknowledgements of all messages (if no acknowledgement of a message is received, the message is sent again). If errors occur (e.g. the message may get lost so the message is sent again, or the reply may get lost so a duplicate message is received). To deal with this, messages are given a unique 'sequence' number to uniquely identify it. This can be used to discard any duplicate messages and keep messages in sequence.

Mutual exclusion

Mutual exclusion is needed to ensure that shared resources are not accessed by multiple nodes simultaneously. Each node is required to keep a queue of pending requests, with the resource being locked in timestamp order. Every request needs to be sent to every other node so that every nodes queue is the same. A node will be able to lock a resource if it's request is first in the queue, and a message has been received from all other nodes with a timestamp greater than the request. This second condition guarantees that no earlier request will still be on its way.

Deadlock

There may be an inconsistent view of the system due to a lack of global time. For example, node A may report it has resource X and is waiting for Y, while node B may report it has resource Y and is waiting for X (these may not relate to the state of the system at the same time). This could be solved using a centralised algorithm (deadlock detector), with messages to the detector consisting of two parts: a table of threads waiting for nodes resources, and a table of resources waited for by nodes threads. These messages can be compared to check consistency. Each message essentially acting as a snapshot of the current system's threads/resources a a particular moment (like version control). Another approach could be to use a decentralised algorithm to create a trade dependancy chain: thread A requests X, but X is held by B. B is waiting for Y, but Y is held by C, C is waiting for etc. etc.

Fault Tolerance

Even with these measures in place, faults can (and do) occur in these circumstances, it's importance to have some error detection/recovery in place. Communication between nodes can fail (i.e no more messages are received, or messages can be garbled). A checksum can be used in these circumstances to verify that the received message is in fact the same as that sent. The nodes themselves can also fail (generating faulty/incorrect messages), to detect this, each node can be compared agains the others.

Distributed Processing

Distributed processing allows processing to be carried out on a remote server. Two varieties considered here:

Remote Procedure Calls

RPC is an inter-process communication that allows a computer program to cause a subroutine or procedure to execute in another address space (commonly on another computer on a shared network) without the programmer explicitly coding the details for this remote interaction. That is, the programmer writes essentially the same code whether the subroutine is local to the executing program, or remote.

The RPC Procedure:

1. The client calls the client stub. The call is a local procedure call, with parameters pushed on to the stack in the normal way.
2. The client stub packs the parameters into a message and makes a system call to send the message. Packing the parameters is called marshalling.
3. The client's local operating system sends the message from the client machine to the server machine.
4. The local operating system on the server machine passes the incoming packets to the server stub.
5. The server stub calls the server procedure. The reply traces the same steps in the reverse direction (server 'unmarshalls' parameters and calls the corresponding procedure).
6. The server returns the results in a message to the client.

Data Representation Issues Associated With RPC:

- Different formats on different machines
- Pointers lose their meaning if sent to a remote machine

To overcome this, the platform-independent External Data Representation (XDR)can be used for common data types. Another solution is to linearise linked data structures.

Remote Method Invocation

RMI is a Java-specific implementation of RPC. There aren't any issues with data representation: any 'serializable' Java objects can be sent. Although Java uses references (pointers) to refer to objects, serialization linearises embedded references automatically. The client accesses 'registry server' to get a remote object that implements a desired interface. The client can call methods of this object as it it were a local object.

To implement, simple extend *Remote*, and add *RemoteException* handling.

Listing 5: Implementing RMI

```
1  /**
2   * The interface
3   */
4  import java.rmi.*;
5  import java.rmi.server.*;
6
7  public class Adder extends UnicastRemoteObject implements Adding
8  {
9      public Adder () throws RemoteException { }
10
11         public int add (int a, int b) throws RemoteException
12         { return a + b; }
13 }
14
15 /**
16  * The server class
17  */
18 import java.rmi.*;
19
20 public class AddServer
21 {
22         public static void main (String args[]) throws Exception
23         {
24                 Naming.rebind ("Adder", new Adder());
25         }
26 }
27
28 /**
29  * The client class
30  */
31 import java.rmi.*;
32
33 public class AddClient
34 {
35     public static void main (String args[]) throws Exception
36         {
37         Adding adder = (Adding)Naming.lookup("//192.168.11.3/Adder");
38                 System.out.println(adder.add(123,456));
39         }
40 }
```

Distributed Object Frameworks

There are a number of higher-level mechanisms for distributing objects in an object-oriented system. Below are some examples.

Common Object Request Broker Architecture

Defined by a consortium including IBM, Apple and Sun, CORBA is designed for use on heterogenous systems. It's similar to RPC, may be different hardware, different languages. CORBA uses an Interface Definition Language (IDL) to define objects in a platform-

independent way. ORB (Object Request Broker) is used to handle communication with remote objects (lookup service returns references to remote objects, a client-side stub/'proxy' marshals parameters etc. Interface repository allows objects to be located by the interfaces they implement

Jini

A Java-specific framework that uses RMI to build federations of services. A lookup service locates objects implementing a particular interface. It returns a proxy object to communicate with the real object, the proxy can perform a mixture of client-side and server-side processing. Access to Jini services is based on leases (a lease entitles you to use the service for a particular period of time). You can be given exclusive access to the service, or non-exclusive access to allow the same resource to be shared concurrently with others.

Proxies Jini uses two styles of proxy: a thin proxy, where everything is passed to the remote object, and a fat proxy where some local processing is performed before passing the request to the remote object. The proxy can validate parameter values before passing them to the remote object, which means more responsiveness at the client end (no waiting for a remote server to tell you that a value is out of range). It also means less loading at the service end (no need to validate parameters sent by the client).

A service can register itself by using the Jini discovery protocol to locate a lookup service, and then use the join protocol to upload a proxy object to the lookup service. If a lookup service is already known then a direct TCP connection can be established, otherwise a multicast UDP request is used to find one.

Networking

Internet fundamentals

TCP/IP

Loosely referred to as 'internet protocols', TCP/IP consists of 5 main layers: hardware, data link (e.g. device driver for Ethernet card), internet (IP), transport (TCP or UDP), application (HTTP etc.). This layered model assumes reliable data transfer at each level; each layer has its own error detection and recovery mechanisms (e.g. by requesting retransmissions). TCP assumes that reliability is an end-to-end problem: individual packets can be lost or corrupted, and rather than each node passing the message from one endpoint to another, only the transport layer deals with errors detection and recovery. TCP assumes that hosts participate in network issues (routing, error handling, network control). Each destination layer receives the object sent by the same source layer (data link layer receives frames sent by the source data link layer etc.).

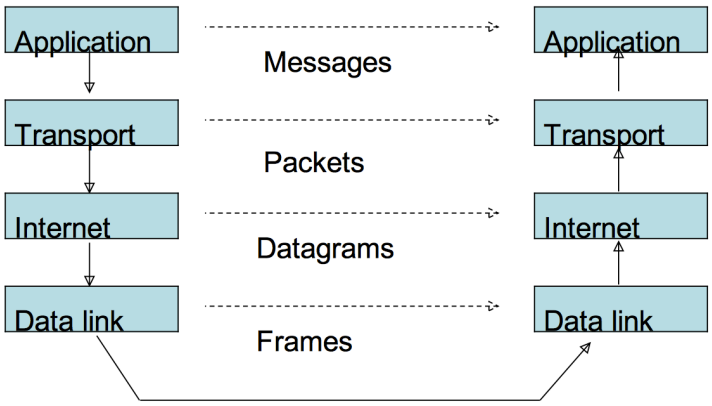

Figure 1: The TCP/IP Layered Structure

Ethernet

Ethernet is a broadcast medium which has become the standard for local area networks. Ethernet stations communicate by sending each other data packets (individual blocks of data). Each Ethernet station is given a 48-bit MAC address to identify it on the network. With ethernet, every frame sent is received by every system connected to the Ethernet (including the transmitter).

Frame format: preamble - alternating 0s and 1s for synchronisation (8 octets), destination (6 octets), source (6 octets), frame type (2 octets), data (64-1500 octets), CRC (4 octets)

Carrier Sense Multiple Access with Collision Detection (CSMA/CD): Used to regulate access: A node waiting to transmits monitors the carrier signal waiting for the line to become idle, at which point it starts transmitting. If another node starts transmitting at the same time, the frame is garbled (detectable by the senders). When a collision is detected, each transmitting node backs off for a random period. This randomness reduces chance of another collision (in the case of further collisions, the maximum period for back-off is doubled - exponential). As the network load increases, collisions become more frequent, causing more nodes to back off. This causes the network to slow down, but should not fail completely.

Internet Addresses

Addresses are IPv4 (or 32-bit), consisting of 4 octets expressed as n.n.n.n (e.g. 127.0.0.1). The next-gen IPv6 (128-bit) consists of 6 octets (giving ∼1038 unique addresses).

Address divided into network and machine number: subnet mask specifies how it is divided

ICMP

Internet Control Message Protocol (ICMP) is one of the core protocols of the Internet Protocol Suite. It is chiefly used by the operating systems of networked computers to send error messages indicating, for example, that a requested service is not available or that a host or router could not be reached. ICMP can also be used to relay query messages. It is assigned protocol number 1. ICMP differs from transport protocols such as TCP and UDP in that it is not typically used to exchange data between systems, nor is it regularly employed by end-user network applications (with the exception of some diagnostic tools like ping and traceroute). ICMP for Internet Protocol version 4 (IPv4) is also known as ICMPv4. IPv6 has a similar protocol, ICMPv6.

RARP

Reverse Address Resolution Protocol is an obsolete computer networking protocol used by a host computer to request its Internet Protocol (IPv4) address from an administrative host, when it has available its Link Layer or hardware address, such as a MAC address. It has been rendered obsolete by the Bootstrap Protocol (BOOTP) and the modern Dynamic Host Configuration Protocol (DHCP), which both support a much greater feature set than RARP.

BOOTP

The Bootstrap Protocol, or BOOTP, is a network protocol used by a network client to obtain an IP address from a configuration server. The BOOTP protocol was originally defined in RFC 951. BOOTP is usually used during the bootstrap process when a computer

is starting up. A BOOTP configuration server assigns an IP address to each client from a pool of addresses. BOOTP uses the User Datagram Protocol (UDP) as a transport on IPv4 networks only.

Historically, BOOTP has also been used for Unix-like diskless workstations to obtain the network location of their boot image in addition to an IP address, and also by enterprises to roll out a pre-configured client (e.g., Windows) installation to newly installed PCs.

Originally requiring the use of a boot floppy disk to establish the initial network connection, manufacturers of network cards later embedded the protocol in the BIOS of the interface cards as well as system boards with on-board network adapters, thus allowing direct network booting.

In 2005, users with an interest in diskless stand-alone media center PCs have shown new interest in this method of booting a Windows operating system.

The Dynamic Host Configuration Protocol (DHCP) is a more advanced protocol for the same purpose and has superseded the use of BOOTP. Most DHCP servers also function as BOOTP servers.

DHCP

Dynamic Host Configuration Protocol is a network configuration protocol for hosts on Internet Protocol (IP) networks. Computers that are connected to IP networks must be configured before they can communicate with other hosts. The most essential information needed is an IP address, and a default route and routing prefix. DHCP eliminates the manual task by a network administrator. It also provides a central database of devices that are connected to the network and eliminates duplicate resource assignments.

In addition to IP addresses, DHCP also provides other configuration information, particularly the IP addresses of local Domain Name Server (DNS), network boot servers, or other service hosts.

DHCP is used for IPv4 as well as IPv6. While both versions serve much the same purpose, the details of the protocol for IPv4 and IPv6 are sufficiently different that they may be considered separate protocols.[1]

Hosts that do not use DHCP for address configuration may still use it to obtain other configuration information. Alternatively, IPv6 hosts may use stateless address autoconfiguration. IPv4 hosts may use link-local addressing to achieve limited local connectivity.

Transport layer

In computer networking, the transport layer or layer 4 provides end-to-end communication services for applications[1] within a layered architecture of network components and protocols. The transport layer provides convenient services such as connection-oriented data stream support, reliability, flow control, and multiplexing.

The most well-known transport protocol is the Transmission Control Protocol (TCP). It lent its name to the title of the entire Internet Protocol Suite, TCP/IP. It is used for connection-oriented transmissions, whereas the connectionless User Datagram Protocol (UDP) is used for simpler messaging transmissions. TCP is the more complex protocol,

due to its stateful design incorporating reliable transmission and data stream services. Other prominent protocols in this group are the Datagram Congestion Control Protocol (DCCP) and the Stream Control Transmission Protocol (SCTP).

Name resolution/DNS

Domain Name System is a hierarchical distributed naming system for computers, services, or any resource connected to the Internet or a private network. It associates various information with domain names assigned to each of the participating entities.

A Domain Name Service resolves queries for domain names (which are easier to understand and utilize when accessing the internet) into IP addresses for the purpose of locating computer services and devices worldwide.

An often-used analogy to explain the Domain Name System is that it serves as the phone book for the Internet by translating human-friendly computer hostnames into IP addresses. For example, the domain name www.example.com translates to the addresses 192.0.43.10 (IPv4) and 2620:0:2d0:200::10 (IPv6).

The Domain Name System distributes the responsibility of assigning domain names and mapping those names to IP addresses by designating authoritative name servers for each domain. Authoritative name servers are assigned to be responsible for their particular domains, and in turn can assign other authoritative name servers for their sub-domains. This mechanism has made the DNS distributed and fault tolerant and has helped avoid the need for a single central register to be continually consulted and updated.

In general, the Domain Name System also stores other types of information, such as the list of mail servers that accept email for a given Internet domain. By providing a worldwide, distributed keyword-based redirection service, the Domain Name System is an essential component of the functionality of the Internet.

Other identifiers such as RFID tags, UPCs, international characters in email addresses and host names, and a variety of other identifiers could all potentially use DNS.

The Domain Name System also specifies the technical functionality of this database service. It defines the DNS protocol, a detailed specification of the data structures and communication exchanges used in DNS, as part of the Internet Protocol Suite.

How is it implemented

Java

Client/server

Many concurrent systems are implemented as client/server systems, where a server provides a service, and clients communicate with the server to use the service (e.g. web servers and web browsers, time servers providing synchronised time). Java comes with the *java.net* package which contains many network-related classes.

TCP/IP

The Transmission Control Protocol over Internet Protocol allows you to connect to a specific 'port' on a remote machine, and read/write data to the port (like writing to a file). TCP is similar to a phone conversation: you connect, communicate and disconnect.

Sockets

A socket connects to a server, and Input/OutputStreams are used to read and write the data.

Socket servers

Socket servers bind to a particular port and wait for someone to connect, at which point it communicates with the client. Server sockets can be set to timeout (*setSoTimeout(5000);*) so that they don't wait indefinitely for a connection.

UDP (Sockets, Packets)

The User Datagram Protocol doesn't carry the same guarantee for communication: a message isn't guaranteed to reach its destination. Data is sent as an individual 'packet', rather than a continuous stream of data. Having sent a request, a response may be sent (or not). As there isn't necessarily a response, the socket should be set to timeout. UDP is more efficient than TCP, and allows for data to be 'multicasted' to multiple recipients. Multicasting allows broadcasting to recipients whose individual addresses aren't necessarily known. UDP is particularly relevant in video streaming for example, where it doesn't matter if odd packets (or frames) are lost.

Real-time

What is a real-time system

All computer systems model some aspect of the outside world, although the timescale doesn't necessarily match that of the real world. Real time systems are required to conform to timescales imposed by the outside world: they must work as 'fast' (or 'slowly') as the outside world.

Hard real time : very tight deadlines; failure to meet deadlines is catastrophic (e.g. aerospace autopilot).

Soft real time : deadlines are more flexible; failure to meet deadlines is not necessarily catastrophic (e.g. multimedia video display applications, financial payroll systems).

Embedded systems

Real time systems are often computer systems which are part of (embedded in) a larger system (e.g. process control, autopilot, manufacturing). Embedded often used as a synonym for real time.

Characteristics

- Needs to cope with a variety of external events (where software is becoming frequently large and complex)
- Reliable and safe: able to detect and recover from failures
- Need to interact with external hardware
- Need to be able to specify timing requirements (when to perform actions, when to complete actions by, what to do when deadlines are missed, importance of granularity (e.g. IBM PC clock granularity is 55ms = 100yds at 600mph)
- Event-driven rather than process-driven (external events must be dealt with as they occur; event ordering is not generally predictable)
- Generally uses concurrent processes (each event source can be handled by a separate process)

- Must have predictable response times (factors to consider: caching/pipelining affect program speed, worst case is an order of magnitude slower than the average, fast programs & external time)
- Code must be safe (i.e. nothing 'bad' will happen) and live (i.e. something 'good' will eventually happen). 'Eventually' must be able to be upper-bounded, as livelock can particularly be an issue in real-time systems.

Security

Password protection

Encryption

Public-key

Steganography

SSL

Types of attack

Trojan

Virus

Worm

Denial of service

Mail bombing

Phishing

Keylogging

Protection