RPC

Distributed Programming Overview

- Client/Server Operations
- Mechanisms for constructing distributed programs
- A distributed system is a set of software components running on a number of computers in a network
- Users interact with application programs
  - Clients request services
  - Servers fill these requests

Remote Procedure Calls (RPC)

- Examples of RPC applications include NFS and NIS
- Inputs are called by value, Outputs are called by reference

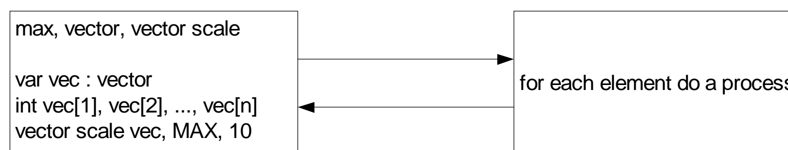
Characteristics

- Call a procedure on a different system
  - Each computer has DIFFERENT ADDRESS SPACE
- RPC
  - Sends a request message
  - Received by the remote program
  - A reply message is sent back

- It has a mechanism to pass pointers – expended data representation (XDR)

RPC to Vector Scale

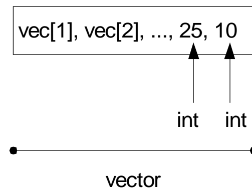
- Global environment containing



VectorScale(vector, 25, 10)

- Arguments are sent (as a request message) in *flattened form*
- Arguments are *unflattened* by the server
- This process is called Mashalling

MAX = 25
Vec[ele1, ele2, …, 25, 10]



- An RPC call has an independent life cycle of the caller
- Failures include
  - A request message is lost
  - A reply message is lost
  - Server crashes and is restarted
  - Client crashes and is restarted
  - Timeouts

RPC Interfaces

- A separate compilation and linking
  - Binding and linking of client and server
- Interface definition
  - A list of procedure names with the types of inputs, outputs and arguments
  - Allows for several languages
- Interface language
  - Specifies services, parameters, procedure names
  - Remote Procedure Language (RPL) allows for
    - Integers, Booleans, Floats, Characters
    - Can define arrays, strings and records
    - Uses scalar data types
- Interface compilers
  - Process definitions
  - Marshall data (in both requests and replies)
  - Dispatching
    - Unique Ids for each process
    - Numbered in order (IE – 0, 1, 2…)

RPC Binding

- Client must locate the server that will execute the request
  - It is not practical to do a bind of the server's host address into the client program at compile time
- Arguments and requests given must conform to those expected by the remote server
  - This is done through a common interface
- As clients and servers are compiled separately and often at different times it is necessary to ensure that they are both compiled within the same RPC interface

- The interface is exported
  - Procedures register – location, port number, checksum and service
  - Procedures withdraw – remove an instance of the server from the *RPC table*
  - whereis – looks up a named service and returns its location

The RPC Software

- Transmits request and reply messages
  - Uses a message passing protocol
- Marshals arguments
  - And in server dispatches calls
- Combines RPC modules with client and server programs (done in conventional languages)

RPC Message Structures

```
TYPE   Message =     Record
                     MessageType : { Request | Reply }
                     RequestID : Integer  /* one per rpc */
                     MessageID : Integer /* one per request message */
                     SRCAddress : Port or Fully Qualified Network Address of the client
                     ProcedureID : Integer
                     Arguments : Flattened list
END
```

Example

```
Write(12345, 100, 5, "Hello")

var m: Message ;
m.MessageType = Request ;
m.RequestID = makeID( ) ;
m.MessageID = makeID( ) ;
m.SRCAddress = 10.0.0.1 ;
m.ProcedureID = 0 ;
m.Arguments = { 12345, 100, 5, "Hello") ;
```

RPC Exchange Protocols

- Request Reply Protocol (RR)
  - Retrieves, timeouts, etc
- Request Protocol (R)
- Request Reply Acknowledge Protocol (RRA)
  - Requires unique Ids for requests
  - Client initiated

Local Procedure Calls

- XDR is an extended data representation
    - file.x – holds definitions of constants, typedefs, enumerations, structs, unions, programs

Example

- remote directory listing
- ls -> rls

jupiter% server &
jupiter% client jupiter /home/mcizkow
File
File1
…

Compiling a RPC Protocol

- Using files:
    - rls.c, rls.x, rls_svc_proc.c, Imakefile
- Run rpcgen rls.x, it creates
  rls.h
  rls_clnt.c – a client stub
  rls_svc.c – a server stub
  rls_xdr.c – Extended Data Representation
- If you get an error about rls.h, the error is NOT in this file, it is in the rls.x file
- Fix the rls.x file and run rpcgen again

RPC Program Numbers

Range	Description
0x00000000 to 0x1FFFFFFF	Defined by Sun
0x20000000 to 0x3FFFFFFF	User Defined
0x40000000 to 0xFFFFFFFF	Reserved

- Give your program a number
    - For known RPC numbers look at /etc/rpc
    - For known protocols look at /usr/include/rpcsvc

Open Network Computer (ONC) RPC Calls

- On the server side
  registerrpc() – server machine
  portmapper() – ready for service and should be registered, waits for incoming connections
- On the client, to use a UDP service
  callrpc() – portmap is then checked out
- Using rpcgen you can specify
  -s Transport (TCP or UDP)

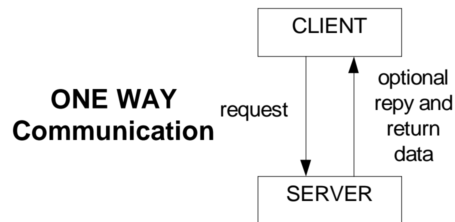
Remote Procedure Call Language (RPCL)

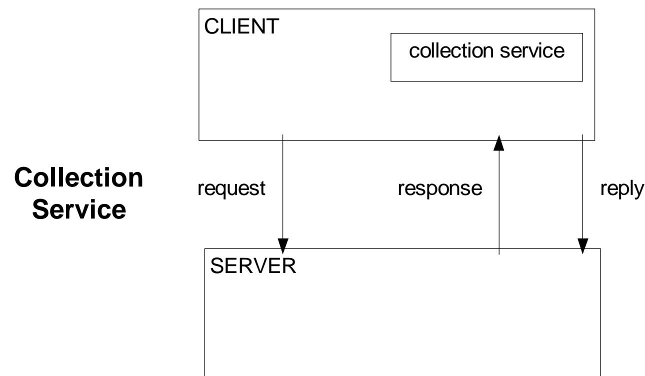
- Used by ONC rpcgen
- An extension of the XDR definition language
- A series of definitions
- We provide a definitions list
- 6 types of definitions
  1. const
     ```
     const MAXSIZE = 8192;
     ```
  2. enum
     ```
     enum colour {
         RED = 0;
         GREEN = 1;
         BLUE = 2;
     };
     ```
  3. struct
     ```
     struct point {
         int x;
         int y;
     };
     ```
  4. union
     ```
     union result switch(int error) {
         case 0:
             opaque data[MAXSIZE];
         default:
             void;
     }
     ```
  5. typedef
     ```
     typedef point polygon[4];
     ```
  6. program
     ```
     program BOGUS {
         version BOGUS_VER {
             void BOGUS_PROC(void) = 1;
         } = 1;
     } = 0x20000001
     ```
- 6 types of declarations
  1. simple
     ```
     colour c;            /* assuming colour has already been typedef’d */
     ```
  2. fixed array
     ```
     colour pallet[8];
     ```
  3. variable array
     ```
     int x <MAXSIZE>;
     int y < >;      /* any size */
     ```

4. pointers
   pt *pNext;
5. boolean
   bool myflag;
6. strings
   string buffer <32>;
   string long < >; /* any size */
7. opaque
   opaque FixedData [512];

- Opaque data is used in XDR and RPC
  - Sequences of arbitrary bytes, with no explicit typing

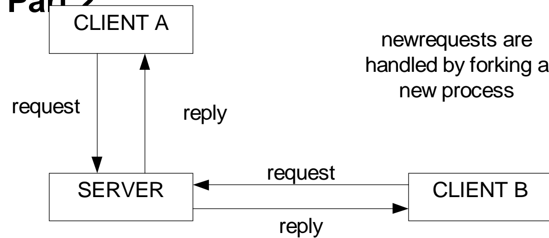
Remote Asynchronous Calls





Multitasking Servers

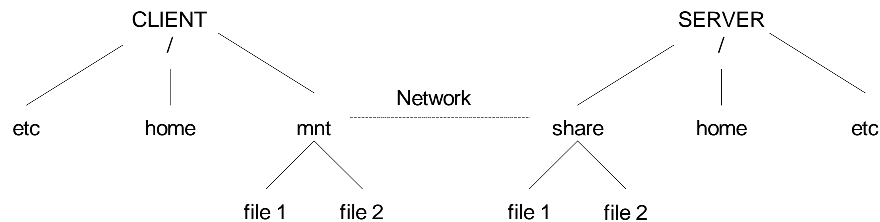
- Want to avoid blocking and return control immediately
  - To do this set timeout to 0 (immediately)
- No client or server listening allowed
- One way requests are determined by the server
- Client requests are done with a call to clnt_control()
- In <sys/time.h> there is a definition for struct timeval
- Use mytime.tv_sec = 0
- Default timeout for RPC is 25 seconds
- Multitasking via multiple processes



RPC Examples

Network File Services (NFS) and Remote File Sharing (RFS)

- Both are distributed file systems that use RPC
- Follow the remote mount model
  - Remote files *look* like they are local



- Goals of RFS include
  - Transparency
  - Preserve full UNIX system semantics on remote files
  - Support any *file type* – IE remote printers, tape drives
  - Support any physical network topology
  - Full data cache consistency
- Availability
  - System Administrator advertises directories that are available for sharing
  - On the server
    server# share –F rfs /usr/local/shared MYSHARE
  - On the client
    client# mount –F rfs MYSHARE /mnt
  - RFS uses a name service which identifies available resources and where they are located
- RFS does not allow for mounting a share which is a share (IE – no multihops)
- Drawbacks
  - UNIX semantics mean UNIX only file systems
  - Can not handle diskless workstations
  - Difficult to setup and maintain
  - Does not handle crashes well

- NFS is a transparent remote file system
- Supports all OS types
- Machine and OS independent
- Heterogeneous file systems
- Everything is accessed like a local file
- Good recovery support
- Portable
- Supports diskless clients

- Shares are done by the Administrator of the server
- Mounts are done by the Administrator of the client
  server# share –F nfs /myshare
  client# mount –F nfs server:/myshare /mnt
- ASIDE: server:/myshare notation is referred to as UUCP nomenclature

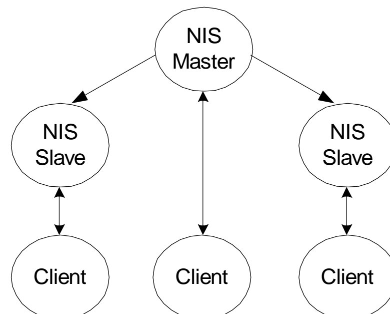
NFS Client Server Interactions

- NFS uses Datagram transport providers (UDP)
- When accessing a remote file, the client obtains a file handle (FH's are associated with the remote file uniquely)
- Client sends a request message to the server requesting a file handle
- Server creates the file handle and passes it back to the client
- This process is an NFS read
- File handles
  - Contain all information for the server to identify the file
  - In SVR4 and R5 the file handle has device and inode numbers
  - Example
    /usr/local/share/myfile1
    has a device number 115
    has an instance number 102
    if myfile1 was deleted the inode number would get reused

Network Information Service (NIS)

- A computing environment with common configuration files (for consistent configuration)
- Uses files including
  /etc/hosts /etc/group
  /etc/ethers
  /etc/bootparams – for diskless workstations
  /etc/services /etc/netgroup
  /etc/aliases /etc/rpc
  /etc/passwd – uses a + to denote a map is required

- All of these files are managed with an NIS database
- In NIS common files are referred to as *maps*
- Servers are referred to as either a *master* or a *slave*



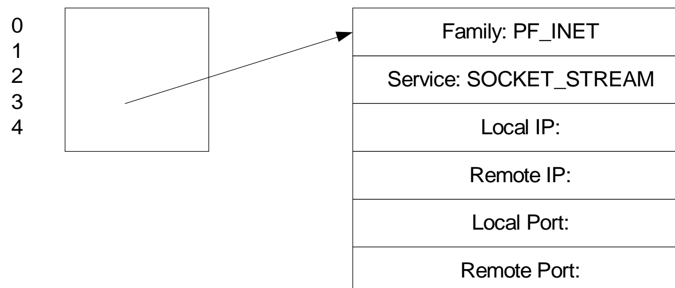
NIS Management

- Installation
  - Build master and slave servers
- Start the services
  - ypserve is a daemon that enables a system to act as an NIS server
- Add clients (slaves)
  - Enable clients by modifying a few files
  - Start ypbind which allows the client to make NIS requests
    # domainname JOE
    # /usr/etc/yp/ypinit –m (map)
    # /usr/etc/yp/ypinit –s (slave)
- The netgroup map is formatted
  hostname, username, domainname
  " – " means that this field takes no value
  source (- , joe, joe), (- , mattc, joe)o
  trusted hosts (chubby, ,)
  trusted users (- , mattc, -)
  dangerous users (- , sean, -)

Berkeley Sockets

Introduction

- A part of the BSD UNIX TCP/IP family (PF_INET)
- A socket descriptor is like a file descriptor
- The OS keeps a descriptor table (one entry per process)



- Sockets that wait for an incoming connection are called *passive sockets*
- Sockets that are used by a client to initiate a connections are called *active sockets*

Protocol Family (PF)	Address Family (AF)	Protocol
PF_INET	AF_INET	Internet
PF_OSI, PF_ISO	AF_OSI, AF_ISO	OSI
PF_LOCAL, PF_UNIX	AF_LOCAL, AF_UNIX	Local UNIX IP
PF_ROUTE	AF_ROUTE	Routing table
N/A	AF_LINK	Link states (ethernet)

- A *pair* refers to an address family and end point address within that family

```
struct sockaddr {            /* struct to hold an address */
    u_short sin.family;      /* type of address */
    u_short sin.port;        /* protocol port number */
    u_long sin.addr;         /* IP address if IP is type */
    char sin.zero;           /* currently unused */
}
```

- Sockets can be UDP or TCP
- From an ls -l you can tell a socket by

```
srwxr-xr-x   2 mattc    users       1024 Mar 12 08:04 test
```

- To send data, use a write call across a TCP connection (send request)
- Read calls are used to receive data across a TCP connection
- Close calls deallocate the socket

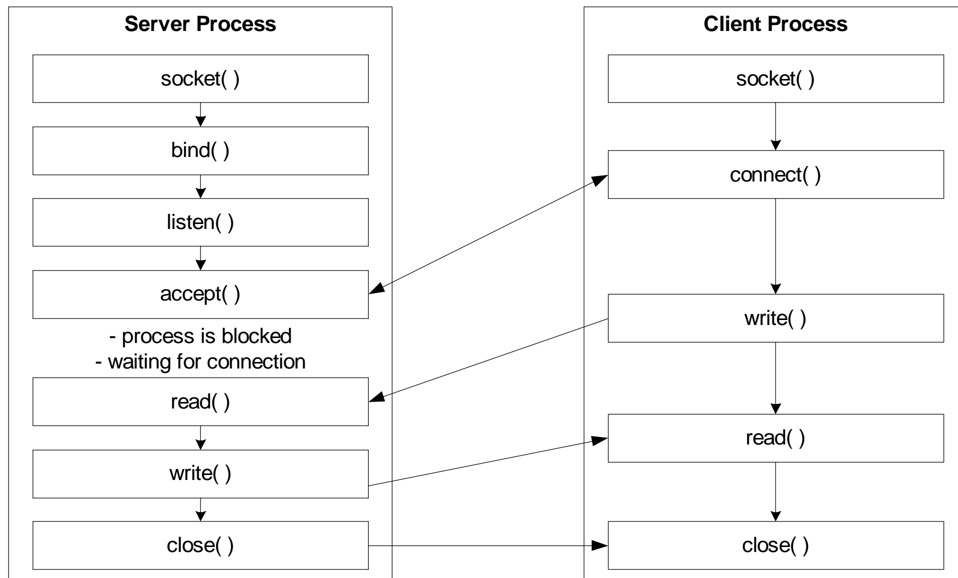
Function Name	Purpose
Socket	To create a descriptor for use in the network
Write	Send outgoing data
Connect	To connect to a remote socket
Read	You guessed it
Close	Terminate and deallocate
Listen	Puts the socket in passive mode
Accept	Accept the next incoming connection
recv / recvmsg	Receive the next incoming datagram
Recfrom	Receive a datagram from a certain endpoint address
Sendto	Send an outgoing datagram
Shutdown	Terminate the TCP connection
Getpeername	Obtain the remote address

Integer Network Order

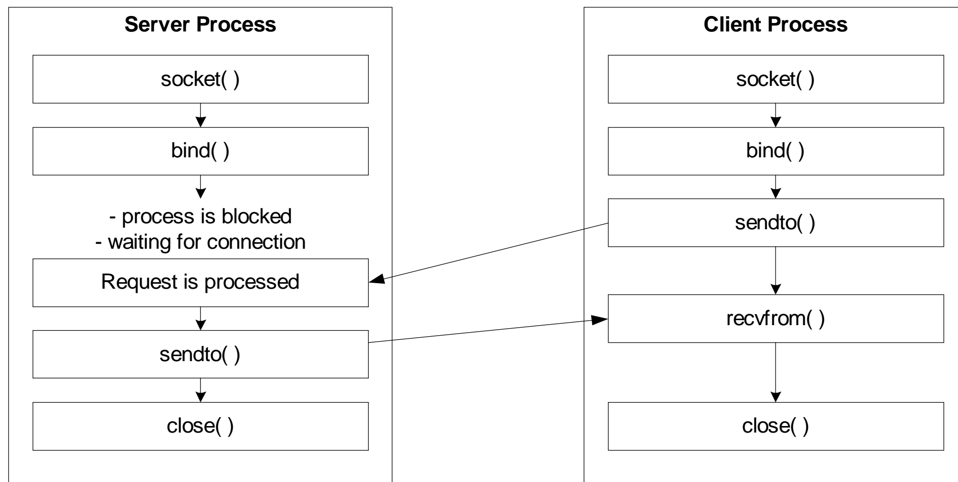
ntoh() – short int, Network to Host
ntohl() – long int
hton() – short int, Host to Network
bcopy() – copy a string, length is specified
bcmp() – compare bytes
bzero() – write a specified number of NULLs to memory
gethostent() – get host entry
gethostbyaddr()
gethostbyname()
gethostname()
getservent()
getservbyport()
getservbyname()

- When using any of the above you have to include <netinet/netdb.h>
- Connection oriented IP addressing uses TCP and you must include <netinet/in.h>

TCP Sockets

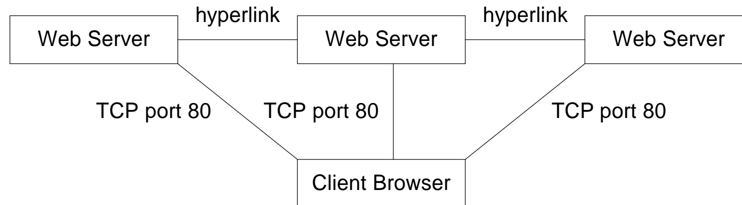


UDP Sockets



Socket Example (HTTP)

- Web Organization



- Uses well known port 80
- Documents are returned to clients in HTML format
- Uses 8 bit ISO Latin 1 character set
- Defines Uniform Resource Locations (URL)
  - Uniform Resource Identifiers (URI)
  - Uniform Resource Names (URN)

HTTP Protocol Definition

- Both requests and responses are considered messages
- Requests consist of
  Request line – request url version
  Headers (0 or more)
  BLANK Line delimiter
  Body
- Three request types
  - GET – returns info identified by the URL
  - HEAD – returns only the headers identified by the URL, used to test validity, accessibility, recent modification
  - POST – sends information through the use of forms
- Format of responses
  Status Line
  Headers (0 or more)
  BLANK Line delimiter
- The status line consists of the HTTP version, response code and response phrase
- Codes are 3 numeric digits and 5 categories, examples include:

1XX	Informational (not in use)
20X	Success
301	Server redirection
40X	Client error
404	Document Not Found
500	Server Error
503	Service temporally out of order

Security

Encryption and Decryption

- Illegal and unauthorized reception of data is referred to as fraud
- Encryption
  - To render information in an *unreadable* form
  - Original text is referred to as plain text (P)
  - Encrypted text is referred to as cipher text (C)

  $C = E_k(P)$ where k is the key and E is the algorithm or function
  $P = D_k(C)$

- A simple encryption scheme is the Caesar Cipher based on character substitution

- Poly-alphabetic encryption changes frequency to destroy common sequences
- The Viginere Cipher is an example

Row 0	A	B	C	…	…	Z
…	…	…	…	…	…	…
…	…	…	…	…	…	…
Row 25	A	B	C	…	…	Z

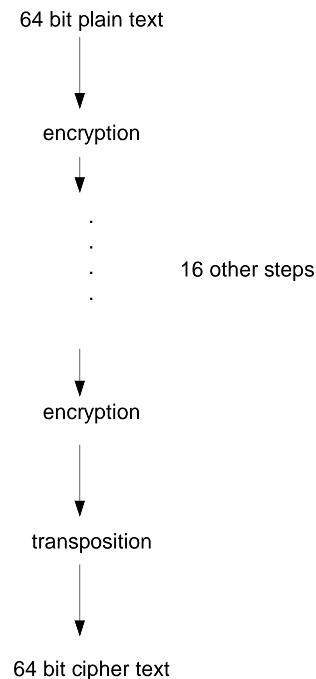
- A letter is replaced with V[(I mod(26), J)
- To replace a letter, let I be its relative position in the message and J be its position in the alphabet, let V be the array

- The Transposition Cipher rearranges a message and stores the plain text in a 2-D array with m-columns
  IE – row 1 has m-letters, row2 has m-letters
- Determine a permutation of numbers, 1 to m and write them as P1, P2, P3… PM
- Then transmit all characters is column 1, column 2, … , column M

- Bit level ciphers encrypt bits like a string
- Divide bytes into *substrings* and encrypt by computing XOR operations between it and the key
- They use the same encryption and decryption algorithm
  (Pi XOR 0) XOR 0 = Pi
  (Pi XOR 1) XOR 1 = Pi
- Security really depends on the length of the encryption key

Data Encryption Standard

- Data Encryption Standard (DES) was developed by IBM in the 1970's
- Uses short keys and complex algorithms
- A government standard in the US and Canada
- DES divides the message into 64 bit blocks and uses 56 bit private keys
- Complex combinations of transpositions, substitutions, XOR operations, etc
- In total there are 19 steps
  - Output of a step is the input to the next

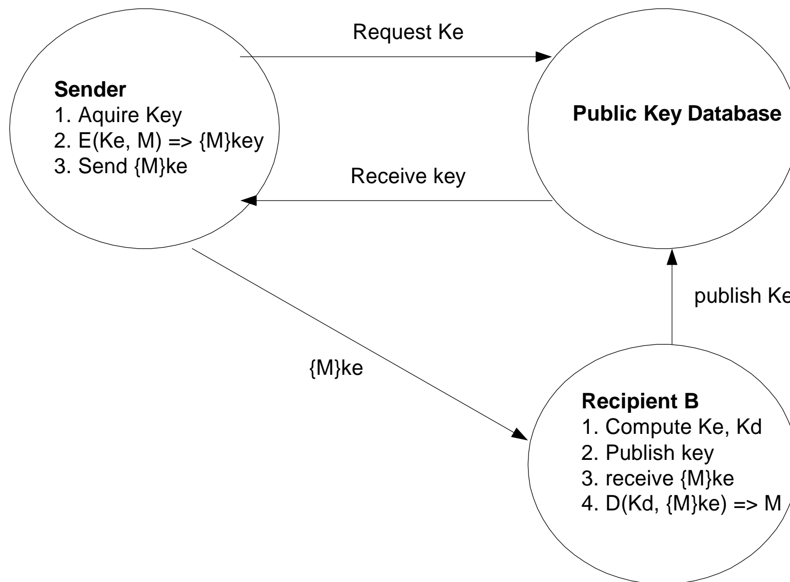


Key Distribution

- For security in general Key Distribution has problems if there is no secure way to distribute a new key
- Merkle's Puzzle Method sends messages containing a potential key and a potential message to try to avoid this problem
- Shamier's Method stores a key so that k people must be present to determine it, it is based on polynomial interpolation

Public Key Encryption

- Public Key Encryption was developed in 1976 by Diffe and Helmann
- Eliminates the need to trust communication between parties to distribute keys
- Works with large prime numbers and is computationally intensive

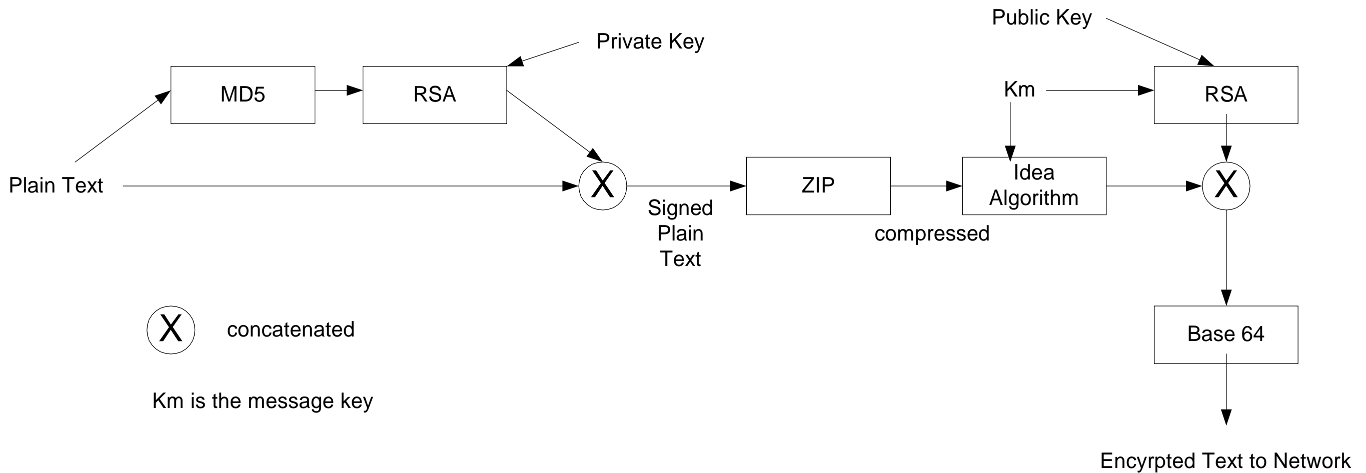


RSA

- Rivest, Shamir and Adelman came up with RSA
- Example: to find a key pair e, d
  1. Choose 2 large prime numbers P & Q (both > 10^{100}) and let N = P * Q and Z = (P – 1) * (Q – 1)
  2. For d, choose a number that is *relatively prime* to Z (no common factors)
     P = 13, Q = 17 so
     N = 221, Z = 192
     So d = 5
  3. Find e from the equation e * d = 1 mod Z
     e * d = 1 mod 192
     385 is divisible by d so e = 385/5 = 77
- To encrypt text, divide the plain text into equal blocks of length 2^k bits
- The function for encrypting a single block of M
  $E(e, N, M) = M^e \bmod N$
- For decryption use
  $D(d, N, e) = e^d \bmod N$

Pretty Good Privacy

- Pretty Good Privacy (PGP) was developed in 1995 by Zimmerman
- It is a FREE package that includes e-mail and digital signature security
- It is debated that PGP infringes on the RSA package



- **Key length**
  - 384 bit is *casual* encryption
  - 512 bit is *commercial* encryption
  - 1024 bit is not yet breakable (April 14, 1999)

- A message M can be Digitally Signed by a principal A by encrypting a copy of M in a key K_a and attached to a plain text copy of M and A's identifier
- Signed documents contain < M, A, {M} K_a > and can be used with secret or public key encryption

Header	Message	
A -> B	M, A	A sends the original message and his signature to B
B -> S	A	B gets A's public key from the server
S -> B	A, $K_{apublic}$	Server supplies the key

B uses the key to decrypt the message from A

- **Digital Watermarks** are embedded into a document (usually graphics) so that if there is fraud the watermark goes away

Kerberos

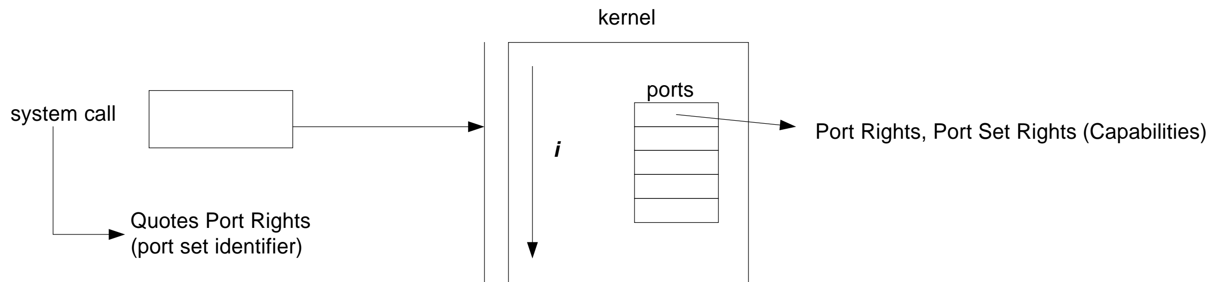
- Kerberos Uses DES
- For NFS, rlogin, password changing and email
- Operation
  - Get *tickets* for kerberos servers (cached in /tmp)
  - Tickets by default expire in 8 hours
  - Tickets are issued by a ticket granting service
  - If 8 hours elapses and a ticket is to be renewed *kinit* with a username and password will renew the ticket
- Services include
  - File services
  - Printers
- Authenticators are special tokens that prove you have the right to a service
- Tickets are cryptographic keys used to create authenticators
- The Ticket Granting System (TGS) is a service that grants tickets
- A session key is a cryptographic key that is used for communication between the user and TGS
  - A *ticket granting ticket* initially gets the session key upon login
- The kerberos server is a master authentication server for the *whole network*
  - it know everyone's password
  - It is usually physically secured
  - It grants the ticket granting ticket
  - Passwords are only used once in initially setting up the session
- *Kerberizing* a system involves setup on each individual workstation
- Steps are as follows:
  - NOTE: Logging in looks the same to the user

Distributed Operating Systems

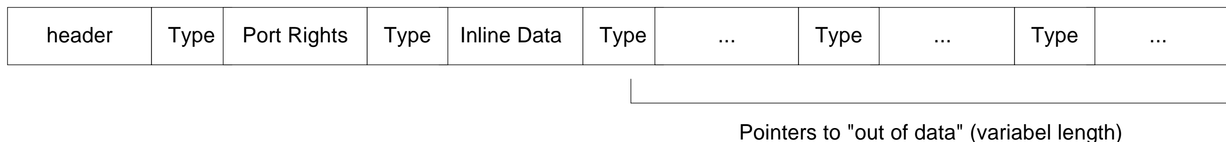
Mach

- Mach is from CMU and is currently at version 2.5 and is being continued by OSF
- Works on Intel, DEC, Sun Sparc, RS6000
- Does emulation for DOS, AIX and OS/2 on top of the Mach micro kernel
- Mach abstractions (terminology) include:
  - Tasks – execution environment, managed by the kernel
  - Threads – execution in parallel on different systems
  - Ports – unicast communication channels with a message queue
    - Handles are assigned to *port rights*
  - Port Sets – a collection of port rights that are local to a task
  - Messages – contain port rights, data and memory management information
  - Memory Objects – virtual address space of a Mach task
  - Memory Cache Object – cache of pages

- Ports and naming
    - A port identifies a resource
    - About 2000 Mach ports defined
    - Port capabilities are like UNIX permissions
        - Send rights belong to a task
        - Ports have receive rights
        - All of this is managed by the kernel which maintains address space for all ports



- New tasks – fork (sort of a blueprint task) on the same computer
- No process/task migration (all tasks stay on the CPU they were created)
- Newly created tasks have no threads
- Threads are created within a task via message passing to ports
- Often stub procedures are used
- Problems are handled by exception ports
- Messages have fixed header sizes and contain pointers to extra data



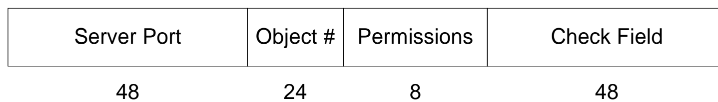
Chorus

- Chorus began in 1979
- Communication processes use *actors*
- A small kernel that *sort of* looks like UNIX
- Micro kernel support for Open System services and emulation of UNIX
- Transparent facilities for networks
- Flexible virtual memory implementation
- Portable (written in C++), modular and machine independent
- Exploits shared memory through the use of dynamically loadable servers
    - Executed at user level or within the kernels address space
- Enhances UNIX with multiple threads and the ability to create processes on remote computers
- Server grouping and reconfigurations are done dynamically
- Distributed memory with multiprocessor operations
- Chorus abstractions include
    - Actors – like Mach tasks
    - Ports – unidirectional communication channels
        - Ports can migrate between actors

- Port Groups – ports can be members of a group (IE – destinations for messages)
- Messages – variable length body (up to 64K) and optionally fixed sized headers (usually 64 bytes)
- Regions, Segments and Local Caches are memory abstractions

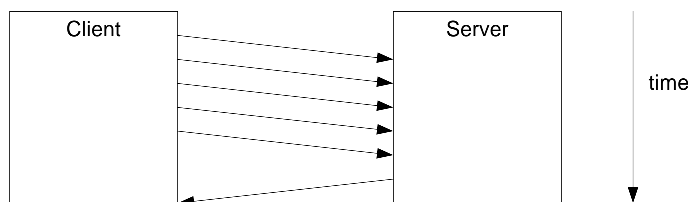
Amoeba

- Amoeba is a completely distributed OS
- Originally by Tanenbeom and currently in version 5.0
- Uses processor pools
- Each resource defines an object
- The micro kernel supports threads and processes
- Amoeba communications use RPC
- Identifiers are mapped at runtime on to a server port
- Object numbers represent objects within the system
- An Amoeba Capability is a request object



- Reduced Capabilities are capabilities with restricted rights

- Uses a multipacket protocol called Fast Local Internet Protocol (FLIP), which is built into the Amoeba design



- FLIP ports for datagram services are IP addresses and a FLIP number identifier
- FLIP ports are looked up by an RPC mapping

Distributed Time and Coordination

- There are various algorithms are used for time coordination and time of day events
- Timestamps are used for serializability
- Security systems use time (IE kerberos)
- Cesium 133 (CS^{133}) is used to give Universal Coordinated Time (UTC)
- UTC is synchronized with broadcasts, at each 5 MHz (AM) interval
- Stations include WWV, WWVH and the Dominion Observatory Canada
- Geostation Environmental Operational Satellite (GEOS)
- Global Positioning System (GPS)
- Compensating for drift
  - Missed ticks mean clocks go out of sync
  - No resetting of time is done, just change the rate of ticks
- Hardware clocks (H) use a compensating factor (δ)
  $S(t) = H(t) + \delta(t)$
  Let $\delta(t) = a(H(t)) + b$ drift is linear
  $\therefore S(t) = (1 + a) H(t) + b$
  T_{skew} when H = h
  T_{real} is actual time
  So $T_{skew} > T_{real}$ or $T_{skew} < T_{real}$
  We want S to have actual time after N ticks so
  Tskew = (1 + a) h + b
  And $T_{real} + N = (1 + a)(h + N) + b$
  Solving:

$$a = \frac{T_{real} - T_{skew}}{N}, \qquad b = T_{skew} - (1 + a) h$$

Cristian's Method

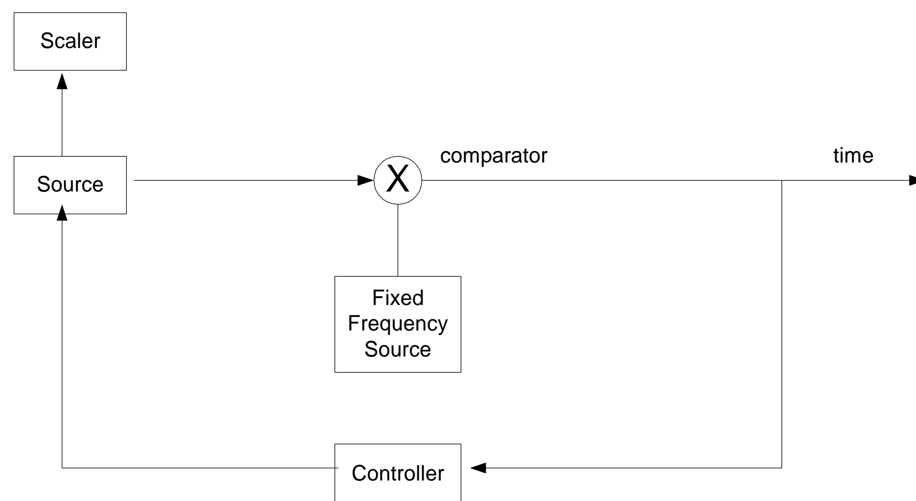
- Cristian's Method for synchronizing clocks uses a server process (S) which supplies time on request
- Subject to variation
- It involves a timeserver using a device for UTC
- A process (P) learns time from S
- Total round trip time from the client to the server is recorded
  $T_{round} = M_r + M_t$
- This time is typically less than 10 ms on a LAN
  $T_{estimate} = T + T_{round}$
- Time for sending and receiving ($M_r + M_t$) are averaged, so that min x is the earliest time
- Range is $[\, T + min, T + T_{round} - min \,]$
- Accuracy is $\pm (T_{round} / 2 - min)$
- Problems with this method include:
  - Timeserver failure
  - Imposture servers

Berkeley Method

- The Berkeley Method for coordinating time involves choosing a *master server*
- The master polls all other computers on the network (*slaves*) and estimates (for each one) an average time and a round trip time
- This value is then used to figure out a Δ t for each slave
- If the master fails, an election on the network occurs and the winner takes over

NTP

- Network Time Protocol (NTP) uses UTC to sync its time
- NTP does frequent resets to compensate for drift
- It provides mechanisms against interference
- Uses average values
- Dynamically reconfigures servers that become unreachable
- Its procedure mode calls look like Cristian's Method
- Uses a phase lock loop (PLL) to readjust



Replication

- To maintain copies of data**,** enhance performance and build fault tolerance
- But… it increases network response time
- Distributed data is often fragmented and put together with binary joins
- Replication takes into account enhanced availability, consistency and allows for data caching

- **Architectural Mode** Replication involves using a *replica manager*
- The manager is a process that maintains replicas and performs operations on them

- In the **GOSSIP Model** replica managers exchange *GOSSIP* about what other workstations are doing
- A primary copy of data is always kept and this is propagated to all other servers
- If the primary fails , a slave takes over

Consistency and Request Ordering

- Consistency and Request Ordering may not necessarily be a function of *update order*
- Constraints are needed to ensure that multithreaded requests are done concurrently
- All processes need to be serialized
- In a replicated environment each replica manager can be considered a *state machine*
- Order of processing requires protocols for ordering
- We require R1 and R2 (two consecutive requests) to be processed in order
- If order is not important, the requests can *commute*
- Casual ordering is sometimes referred to as *happening before*
- Some processes must be totally in order, this is referred to as *sync ordering*
- A synchronizer may be needed to sequentially process data

- If an event A happens before event B a method is needed for concurrency control and deadlock prevention
- This method is typically implemented by **Timestamping**
- When using timestamps a predetermined order is needed
  TS(A) uniquely identifies A
  For 2 events, if A occurs before B then TS(A) < TS(B)
- If A happens before B and B before C then A must happen before C
- A, B and C have a transitive relationship
- Processes must be made serializable to solve read and write issues
- Let $R_i(X)$ and $W_i(X)$ denote read and write operations
- Let T_i be a transaction
- A schedule is needed such that
  S1: $R_i(X)$ $R_j(X)$ $W_i(X)$ $W_j(X)$ does not occur
- In the above writing J would overwrite whatever writing I accomplished

- If a transaction can perform operations to data at many sites a method for **Serializing Distributed Databases** is required
- Each site must implement a *local schedule*
- The execution of N distributed transactions may occur at M sites
- A set of master schedules is needed to ensure consistency
- In a distributed database environment **Two Phase Locking** is generally used as a method for concurrency control

- If site 1 is waiting for site 2 and site 2 is waiting for site 1 this is a **Distributed Deadlock**
- Assume each transaction reads X and increments it, then writes the new value, does the same to Y
- If T_i and T_j are activated almost simultaneously 2 phase locking will ensure serializability
- T_i will happen before T_j or T_j will happen before T_I

Sample XDR file

```
typedef string nametype<MAXNAMELEN>;
typedef struct namenode *namelist;

struct namenode {
        nametype name;
        namelist pNext;
};

union readdir_res switch (int errno) {
case 0:
        namelist list;
default:
        void;
};

program FOLDPROG {
        version FOLDVERS {
                readdir_res
                READDIR(nametype) = 1;
        } = 1;
} = 0x20009713;
```

Sample Client

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <string.h>
#include <Xm/List.h>
#include <X11/Intrinsic.h>
#include <Xm/Xm.h>
#include "myrpcfold.h"

extern int errno;

// THIS is basically RLS with adding to a widget
rpcfolders(char *dir, Widget *folder_list) {
  XmString folders, tmp;
  int filecount = 0;
  CLIENT *cl;
  char *server = "jupiter.scs.ryerson.ca";
  readdir_res *result;
  namelist nl;

  cl = clnt_create(server, FOLDPROG, FOLDVERS, "tcp");
  if (cl == NULL) {
    clnt_pcreateerror(server);
    exit(1);
  }
  result = readdir_1(&dir, cl);
  if (result == NULL) {
    clnt_perror(cl, server);
    exit(1);
  }
  if (result->errno != 0) {
    errno = result->errno;
    perror(dir);
    exit(1);
  }
  for (nl = result->readdir_res_u.list; nl != NULL; nl = nl->pNext) {
    if ((!strcmp(nl->name, ".") || !strcmp(nl->name, "..")) != 1) {
      folders = XmStringCreateLocalized(nl->name);
      filecount = filecount++;
      XmListAddItemUnselected(folder_list, folders, filecount);
   }
  }
  XmListAddItemUnselected(folder_list, XmStringCreateLocalized("inbox"), 1);
}
```

Sample Server

```
#include <rpc/rpc.h>
#include <sys/dirent.h>
#include "myrpcfold.h"

// THIS IS UNCHANGED FROM THE ORIGINAL RLS_PROC - given in class
extern int errno;
extern char *malloc();
extern char *strdup();

readdir_res *readdir_1(nametype *dirname) {
        namelist nl;
        namelist *nlp;
        static readdir_res res;
        static struct dirent *dirp = NULL;
        struct dirent *d;

        dirp = opendir(*dirname);
        if (dirp == NULL) {
                res.errno = 1;
                return(&res);
        }

        nlp = &res.readdir_res_u.list;
        while (d = readdir(dirp)) {
                nl = *nlp = (namenode *)malloc(sizeof(namenode));
                nl->name = strdup(d->d_name);
                nlp = &nl->pNext;
        }
        *nlp = NULL;

        res.errno = 0;
        closedir(dirp);
        return(&res);
}
```

Sample Client

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <string.h>
#include <Xm/List.h>
#include <X11/Intrinsic.h>
#include <Xm/Xm.h>
#include <Xm/Text.h>
#include <Xm/TextF.h>
#define DIRSIZE     8192

#include "host.h"
#include "port.h"

messageclnt(char *command, Widget message_text) {
  char    dir[DIRSIZE];
  int     sd;
  struct sockaddr_in sin;
  struct sockaddr_in pin;
  struct hostent *hp;
  char welcomemsg[100], junk_one[100];
  char response[DIRSIZE];
  XmTextPosition pos = 0;;

  if ((hp = gethostbyname(HOST)) == 0) {
    perror("gethostbyname");
    exit(1);
  }

  /* fill in the socket structure with host information */
  memset(&pin, 0, sizeof(pin));
  pin.sin_family = AF_INET;
  pin.sin_addr.s_addr = ((struct in_addr *)(hp->h_addr))->s_addr;
  pin.sin_port = htons(MYPORT);

  /* grab an Internet domain socket */
  if ((sd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
    perror("socket");
    exit(1);
  }

  /* connect to PORT on HOST */
  if (connect(sd, &pin, sizeof(pin)) == -1) {
    perror("connect");
    exit(1);
  }
  if (recv(sd, welcomemsg, DIRSIZE, 0) == -1) {
    perror("recv");
    exit(1);
  }
```

```
/* send a message to the server PORT on machine HOST */
  if (send(sd, "message", 20, 0) == -1) {
    perror("send");
    exit(1);
  }
  if (recv(sd, junk_one, DIRSIZE, 0) == -1) {
    perror("recv");
    exit(1);
  }

  if (send(sd, command, strlen(command), 0) == -1) {
    perror("send");
    exit(1);
  }
  /* wait for a message to come back from the server */
  if (recv(sd, response, MSGSIZE, 0) == -1) {
    perror("recv");
    exit(1);
  }

  XmTextSetString(message_text, "");
  XmTextReplace (message_text, pos, pos, response);
  close(sd);
}
```

Sample Server

```
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <sys/wait.h>
#include <dirent.h>
#include "port.h"

#define BACKLOG 10     /* how many pending connections queue will hold */

main()
{
  // Socket Specific variables
  int sockfd, new_fd;  // listen on sock_fd, new connection on new_fd
  struct sockaddr_in my_addr;    // my address information
  struct sockaddr_in their_addr; // connector's address information
  int sin_size;

  char *welcomemsg = "Welcome to Matt's Server\n"; // when a connection is made
  char function[100]; // which function to use

  // in all functions
  char response[150];  // first question?
  char command[100];  // what goes to the command line
  char result[600];  // used to return some of the results
  int i;  // for counting
  char buf[BUFSIZ]; // for terminating strings
  char cmd[200];  // the command that is exec'd with command as a parameter
  FILE *pp, *popen(); // in used for exec's

  // Variables for FOLDER function
  char *maildir; // for ls /home/mciszkow/mail
  int bytes_rcd; // for terminating string - not 100% necessary
  DIR *dirp;
  struct dirent *dp; // used in the ls type funtion

  // Variables for MESSAGE - MSGSIZE is in PORT.H
  char msgres[MSGSIZE];
```

```
  if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
    perror("socket");
    exit(1);
  }
  my_addr.sin_family = AF_INET;         /* host byte order */
  my_addr.sin_port = htons(MYPORT);     /* short, network byte order */
  my_addr.sin_addr.s_addr = INADDR_ANY; /* auto-fill with my IP */
  bzero(&(my_addr.sin_zero), 8);        /* zero the rest of the struct */

  if (bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr)) \
      == -1) {
    perror("bind");
    exit(1);
  }

  if (listen(sockfd, BACKLOG) == -1) {
    perror("listen");
    exit(1);
  }

  while(1) {  /* main accept() loop */
    sin_size = sizeof(struct sockaddr_in);
    if ((new_fd = accept(sockfd, (struct sockaddr *)&their_addr, \
                         &sin_size)) == -1) {
      perror("accept");
      continue;
    }
    if (!fork()) { /* this is the child process */
      if (send(new_fd, welcomemsg, strlen(welcomemsg), 0) == -1)
        perror("send");
      if (recv(new_fd, function, sizeof(function), 0) == -1) {
        perror("recv");
        close(new_fd);
      }

// MESSAGE - WORKS
      else if(!strncmp(function, "message", 7)) {
         sprintf(response, "For which folder and number\n");
         if (send(new_fd, response, strlen(response), 0) == -1) {
           perror("send");
         }

         bytes_rcd = recv(new_fd, command, sizeof(command), 0);
         if (bytes_rcd == -1) {
           perror("recv");
         }
         command[bytes_rcd] = '\0'; // NULL TERMINATE the string

         sprintf(cmd, "readmsg -f %s", command);
         if (!(pp = popen (cmd, "r"))){
           perror (cmd);
         }
```

```
        msgres[0] = '\0';
        while(fgets(buf, BUFSIZ, pp) !=NULL) {
          for (i=0; i < BUFSIZ && buf[i] != '\n'; i++)
            ;
          sprintf(msgres, "%s%s", msgres, buf);

        }
        fclose(pp);

        if (send(new_fd, msgres, strlen(msgres), 0) == -1) {
          perror("send");
        }
      }
}
```