Remote Device Model

Introduction

As demands on embedded systems grow in sophistication
So does the complexity of their design
Here-to-fore tasks doing majority of the work in the application
Have been primarily found in main processor.

Local device model focuses on and supported interaction with peripheral devices
In close proximity to local system
Any tasks executing on such devices
Associated with function of specific device
Generally not considered part of the application proper

The other component of external world interaction
Introduces
  *Remote device model*
  *Notion of distributed tasks*

  Such tasks often considered to be contributing part of main application

  We now add remote capability to expanding embedded system architecture

Remote vs. Local

Major differences between the remote and local device models
Reflected in details of
  *Control and synchronization of the information exchange*
  *Transport mechanism*

  In local model interprocess control, communication, and synchronization
  Primarily depended upon number of variations on shared variable paradigm

  Local device model works well as long as peripheral devices
Within three to five meters of controlling microprocessor

Frequently data movement done in parallel
  Far more efficient at moving large numbers of bits than serial
  Yet not always as practical

With remote device model as we expand to greater distances
  Thinking shifts to network based approach
  Tasks often exchange information through messages

  Sending process transmits a set of data values - message
  Through specified communication medium
  Where it is accepted by receiving process

Within a widely distributed embedded system
  Most remote intra and inter system communications
  Takes place over a standard network using a serial scheme
  EIA-232, I²C, SPI, CAN, Ethernet, USB, etc.

Any modern automobile provides an excellent example
  Processors throughout vehicle manage everything
  Fuel system
  Passenger environment and entertainment systems

Internet and Internet appliances
  Further examples of contemporary distributed embedded applications

Under both remote and local models
  One of the basic goals of distributed system
  Ensure that the underlying architecture
  Invisible to tasks comprising application

From any task’s perspective
  Interaction with other tasks should not depend upon
  Where the tasks are physically located
  Computing engine on which each is executing

  We want to be able to easily and seamlessly exchange information
  With any part of the system
  Notion of highly cohesive loosely coupled modules continues

The Remote Model – A Quick Look
  Remote model will follow that set for local model
  Constituents
  - Information
  - Places
• Control and synchronization
• Transport

Look first at new challenges with remote model architecture

Challenges
Very nature of distributed systems introduces new challenges
• Possibility of local failures now exists
  System can experience hardware or software failures
  Of or on any of the distributed portions of the application
  While remainder of the system continues to operate

  The designs must be tolerant of such failures

• Complexity of interprocess communication and synchronization increases
• Communication delays become longer
  Have the potential to become nondeterministic
  Greater impact on system timing
• Need to meet hard real-time constraints remains
  Increasing the complexity of any analysis and modeling of such problems

The Pieces
Places and Information
  Earlier diagram gave a high-level architecture for remote device model
  Data content of information exchanged
  Varies little from that in local model

  Content may now include control and synchronization information

  Embodied in messages exchanged over the remote network structure
  Identity of places
  Where information is to be written to or read from
  The information itself

  Format of the message can be implemented in a variety of ways
  Depending upon
  Nature and structure of the underlying protocol
  Supporting networks.

Control and Synchronization
  The control and synchronization strategy
  Incorporated into the protocol by which messages are exchanged
Transport

The physical transport of the information between System core Remote external devices Can utilize any of the models discussed with local model

Copper wire remains the medium of choice However, fiber and air gaining wider spread support Within programmable logic devices or networks on a chip It’s a silicon path

Implementing the Remote Device Model – A First Step

Protocol for transmitting and receiving information or message Central to any such communications between electronic devices

Transport and Control Component

Will begin discussion of remote model with the transport component Necessarily must also include control aspects

As noted - Message exchange in distributed embedded applications occurs via either

- Proprietary network
- One implemented according to one of the many standards

Generally transport topology is serial and information flow is full duplex Whether it utilizes a proprietary or standard topology

Typical transport architecture comprises a hierarchy of virtual networks

Above physical portion of the transport mechanism We have a varying number of software layers or levels as illustrated
Function at each level or layer on one machine
   Interacts with the corresponding function at same level on second machine

At each level potentially a different language spoken
   Referred to as a protocol

Function at each level
   • Provide services for the level above
   • Uses services of level below

Between levels have relationship of
   Service provider and a service consumer

At each level protocols may be implemented in either hardware or software
   Generally lower levels are done in hardware and upper in software

Terminology
   Entire collection
      Called network architecture
   Set of protocols used
      Called a protocol stack
   Information sent on each level
      Called a message
   Possible that a message on higher level
      Composed of several lower level messages

Synchronization between or among distributed processes
   Accomplished through such a message exchange

Today, a wide variety of protocol standards are available
   Although there are times when a proprietary network and protocol must be used
      Given a choice one should opt for one of the standards

General objective of the standards
   Facilitate message exchange in a specific application context such as
      Small computer networks (EIA-232 or USB)
      Simple local area networks (Firewire, Bluetooth, I^2C)
      Automotive networks (CAN bus)
      Manufacturing environments (CAMAC)

While unique to their particular context
   Most of these models trace back to two major protocol schemes or stacks
      OSI and TCP/IP
The OSI and TCP/IP Protocol Stacks

OSI *Open Systems Interconnection model*
Proposed and developed by International Standards Organization – ISO
OSI protocol specifies a 7-layer virtual machine

TCP/IP *Transmission Control Protocol / Internet Protocol*
Comprises 5-layer virtual machine

Physical and data link layers of OSI
Combined into host to network layer in TCP/IP

Following diagrams present and compare
Hierarchical architecture and layers
For the OSI and the TCP/IP models

At the network layer and below
Models are hardware based
Above that level
Models expressed in software

<table>
<thead>
<tr>
<th>OSI</th>
<th>TCP / IP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Physical</td>
<td>Host to Network</td>
</tr>
<tr>
<td>Data Link</td>
<td>Internet</td>
</tr>
<tr>
<td>Network</td>
<td>Transport</td>
</tr>
<tr>
<td>Transport</td>
<td>Not Present</td>
</tr>
<tr>
<td>Session</td>
<td>Not Present</td>
</tr>
<tr>
<td>Presentation</td>
<td></td>
</tr>
</tbody>
</table>

![OSI and TCP/IP Protocol Stacks Diagram]

 physical network layer, data link layer, physical layer, application layer, presentation layer, session layer, transport layer, network layer.
The Physical Network

Similar in both OSI and TCP/IP models
Can be implemented as
    Copper, fiber, or air connection

Provides physical means to move collections of bits (1's and 0's)
Over a communications channel
No meaning or structure to the collections

OSI – Physical Layer
At this level
Concern is for
  ✓ Mechanical and electrical interfaces
  ✓ Integrity of bits
  ✓ Physical characteristics of the bits
    Logical 1 or 0
    Such characteristics include
      Number of volts
      Width (in time) of each bit
      Determines bit transfer rate through the channel

  ✓ Transfer scheme
    Uni-directional
    Bi-directional
  ✓ Connectors and cabling

Control issues address
  How a connection is established and released

OSI – Data Link Layer
Moves collections of bits aggregated as frames
Sender
  Breaks the data stream into data frame
Receiver
  Acknowledges reception via an acknowledgment frame

Data Link Layer must
  ✓ Create and recognize frame boundaries
    Accomplished by surrounding a frame with delimiters
      Header and trailer
    Delimiters must be distinguishable from the data frame
  ✓ Handle frame re-transmission in the case of corruption
    Must accommodate duplicate re-transmission
    If acknowledge frame is lost
  ✓ Handle duplex transmission and acknowledgment
  ✓ Handle broadcast traffic including shared channel access
TCP/IP – Host to Network

No significant requirements are specified at this level
Host system must simply be able to
Connect to the network
Transmit or receive IP (Internet protocol) packets

The OSI network layer corresponds to the TCP/IP Internet layer.

OSI – Network Layer

Manages the routing of a transmission from the source to the destination

Routing alternatives
  Fixed path via static tables
    Predetermined and integrated into the network
  Determined at the time of the connection
  Dynamically modified throughout the session
    May be done to relieve congestion
    Accommodate network failure
    Find ‘shortest’ path

Must accommodate
  Different characteristics between or among networks
    Addressing
    Message size
    Protocols

Manage accounting for network use

At the network layer and below
  Activities are directed toward
    Managing the network
    Physical movement of data
  Bits are collected into manageable packets

Above the network layer
  Collection of virtual machines
    These have the task of managing the session
    The top-level message is broken down into manageable packets

The interface occurs at the transport layer
  Below is the subnet
  Above is the session

TCP/IP – Internet Layer

As noted
  Key layer
Define the official
  Packet format
  Protocol
  IP - Internet Protocol
Objective
Get message comprised of packets
From point A to point B
No requirement placed on
Packet ordering during transmission
Route a packet may take
All packets may not take the same route

OSI - Transport Layer
Among the tasks of the transport layer
Accept data from the session layer
Immediately above
Subdivide into packets that are compatible with the network layer
Immediately below
Desire transactions to be implemented such that
Hardware appears invisible to the higher layers
Must accommodate
Speed demands
High speed
Subdivide problem into multiple parallel transmissions
Low speed
Implement as a single path
Price demands
Maximally use the transport mechanism is the transaction cost is high
Determine the type of service
Point to point in order sent
Transport of individual message components
Broadcast to multiple destinations
Control the flow of information
Prevent over / under run

TCP/IP - Transport Layer
Equivalent to the OSI transport layer
Similar responsibilities
Two communication protocols are defined for the layer
TCP
UDP
TCP
TCP - Transmission Control Protocol
Very reliable
Connection oriented protocol
Ensures data stream
Originating on one machine
Delivered to any other machine on the internet
Process
Disassembles the byte stream into packets
Passes them to the internet level
Handles flow control

UDP
UDP - User Datagram Protocol
Unreliable
In the sense that messages are not inherently acknowledged
Connectionless protocol
Alternative to TCP
Designed for hosts who want to implement their own
Packet sequencing
Flow control
Finds application in
Request - response type applications
Client server
Trade speed for accuracy

OSI - Session Layer
Permits users on separate or different machines to communicate
Like transport layer - supports movement of data between machines
Offers richer set of features and capabilities
Analogous to moving to a higher-level language
Can do a job in assembler
Easier to do in language like C

Potential services
Note
Moving from must dos to offers to do
Manage dialog control
For single direction transmission
Track turns to send
Manage tokens in token passing protocols
Example
IBM token ring
Like a relay race
Cannot use the network until you have been given permission
Permission is in form of a token passed around
Synchronize transactions
Reassemble message if necessary
If complete transfer cannot be completed in single session
Major error
Line drop
Machine crash
Rather than retransmit complete message
Resume from point of last correct reception

OSI – Presentation Layer
Moving up another layer to richer set of tools
Goal of presentation layer is to offer generic set of solutions to common problems
Potential services
  Map information - types, structures, encoding etc.
  From source computer representation
  To network representation
  From network representation to destination representation
Similar to p-code generated from some compilers
  At the p-code level can be moved between machines
  At the machine code level cannot be moved

OSI – Application Layer
This level also deals with incompatibilities between
  Systems at opposite ends of the network
    Hardware
    Some accommodation at this level
    Usually done at a lower level
  Software
    Primary focus
Potential incompatibilities
  File systems
  Terminal types
  Mail systems
  Remote procedure execution

TCP/IP – Application Layer
Supports / Contains all the high level protocols
Three basic ones in support of the original intent of the development
  Virtual Terminals - TELNET
    Recall the discussions of OSI applications layer
  File Transfer - FTP
    File transfer protocol
  Electronic Mail - SMTP
    Simple mail transfer protocol
Later additions
  Mapping of host name to network address - DNS
    Domain Name Service
  Moving news articles - NNTP
    Network news transfer protocol
  Interface to the WWW - HTTP
    Hypertext transfer protocol
The Exchange Models

Most distributed embedded systems implement a message-based exchange

Utilizing any of three patterns of communication, control, and synchronization

- **Client - Server**
- **Peer-to-Peer**
- **Group Multicast**

Will develop and study each pattern in greater depth

- Introduce individual patterns
- Identifying major attributes

Client-Server

Start with *Client – Server* model

Client-Server oriented towards service provision

Pattern involves

- Two messages
- Specific form of synchronization

Exchange flow of control

1. Transmission of request from client process to server process
2. Execution of request by server
3. Transmission of reply to client

Server process

Aware of message as soon as it arrives

Activity in sending (client) process blocks or suspends

Until reply received

Provides form of synchronization

Process commonly represented at language level as

*Remote procedure call*

Hides underlying communication operations

At logical or functional level

Exchange appears to be directly between client and server process

Thereby supports notion of transparency

In reality interchange managed by local kernel

Signals move

- From or to respective software drivers
- To and from the physical network
Peer to Peer Model

The peer to peer model follows naturally from Client – Server model

In the model there are several (peer or equal) processes
Co-operating to
✓ Solve a problem
✓ Share information

Notion of a pre-designated client or server does not exist
Any member of the network may
Request a service
Provide a service
From or to any of the others

Approach can remove potential bottleneck that arises in client-server model
When several clients must interact with single server at same time

Peer to peer model permits several nodes to provide requisite services

As in client-server model communication, control, and synchronization
Achieved through the message exchange

A portion of the architecture of such a system appears in accompanying figure
As minor modification client server model

Group Multicast

In Group Multicast model
Processes interact via message passing
• Single sender
• Multiple receivers

Motivation
✓ Locate an object
  Multicast name of file directory
  To group of server processes
  One holding directory responds

✓ Fault Tolerance
  Multicast same task to multiple servers
  If one fails process can continue

✓ Multiple update
  Information can be Multicast to group of interested processes

Multicast not mutually exclusive
With other communication and synchronization patterns
In a peer to peer network
Multicast might be used by a node entering network
• To announce its presence
• Services that it can provide

Implementing the Remote Device Model – A Second Step
Now examine the communication / control schemes in greater detail
By looking at fundamental components of such systems
Include
• Messages
• Underlying data structures

Important to note
Message may comprise three aspects of Remote Device Model
• Bits identifying the Places
• Bits expressing Information
• Bits supporting Control and Synchronization

The Messages
When designing and implementing distributed embedded systems
Quickly discover that remote operations comprise a substantial proportion
Interactions between processes

Such operations initiated by
One process sending a request message to other process(es)
Receiving process(es) responding with an acknowledgement or a reply message
Indicating
✔ Returning status information
✔ Requested operation has been or will be carried out
✔ Returning results of requested operation

Viewed from most abstract level
Message is simply collection of bits
How bits or groups of bits are interpreted is key

Exchange is controlled movement of those bits from one place to another

To make design of message based communication tractable
Rules are applied to specify
• Exchange
• Interpretation of bits

The Message Structure
One interpretation of collection of bits
Some are header information
Some of the bits are *data* or *payload*.

Header information facilitates job of moving message
- Added by communication driver
- Provides *addressing* (places) and *control and synchronization* information

Payload is the *information* being transported
- Moving payload from one place to another
- Ultimate objective of sending the message

Not all messages are same size
- Simple
  - Occupying only a few words of memory
- Complex
  - Comprising a large number of blocks of data

Design of the exchange process is cleaner and more robust
- If organization ensures fixed sized groupings always transferred
  - Such groupings variously referred to as *datagrams* or *packets*
  - As shown in accompanying diagram

Occasionally padding or fill bits included
- Ensures packets are proper size
- If insufficient data available to complete packet

**Message Control and Synchronization**
- Two kinds of control and synchronization in message based exchange
  - Header information
  - Scheme for managing the data transfer

**The Header**
- Header information is overhead
  - Necessary for getting the data from one place to another
- Added by the communication driver
  - At each level within the protocol stack
  - Based upon the requirements of the exchange protocol
  - At that level

Potential header elements might include
- Destination address or message identifier information
  - At minimum identifies the destination for message
- Routing information
- Identify both the sender and receiver of message
- Indication of the size of the message
  - Done several ways
Unique start and end identifiers
Start identifier and a length field
- Error management information
- Information about message type or structure
  May be desirable to distinguish between
  Data type messages and command type messages

The Transfer Scheme – The Control and Synchronization
Physical information transfer may follow
Proprietary protocol
One of the standards or derivatives discussed earlier.

Within such protocols two kinds of services are identified
Connection oriented
Connectionless

Connection Oriented
Connection oriented service
- Connection established between source and destination
  Prior to the exchange of any data
- Exchange follows
- Connection is terminated

Messages enter one end and are extracted from the other end
Ordering is preserved
Exchange is designated as reliable

Scheme is referred to as circuit switching
Each packet will be sent
In turn through the same physical path

Connectionless Service
Connectionless service
Does not establish a specific connection prior to the start of exchange
Each packet
- Carries full address information
- May arrive at the destination through any of several different routes
  Address information has been added to each packet
- May not arrive in the same order in which it was sent

Such a scheme referred to as packet switching
Exchange is designated as best effort
Messages may be sent as
Datagram service
Message is sent but the receiver does not acknowledge

Acknowledged datagram service
Message sent and the receiver acknowledges

Request-reply service
Sender transmits a datagram containing a request
Receiver returns a datagram containing the answer

Latter scheme is often used in the client-server model

Working with the Remote Device Model – Remote Tasks

Preliminary Thoughts on Working with Remote Tasks
We begin with some thoughts and considerations
When working with remote tasks

We’ll start with the client-server model
With this model we have two kinds of processes
✓ Servers - Provide a service
✓ Clients - Request a service

Our embedded application may be
Server
Client
Both

Important to be aware of several major differences
Between local and remote tasks

Starting at a high level
Initial view of distributed client-server interaction
Presented in accompanying block diagram

Client and server each assume direct communication with the other
Communication link proceeds
Through local kernel - client or server side
To the network
To remote node - server or client side
   Where the message is
       Interpreted
       Passed to that local process

At this level
   Implementation seems rather straightforward
   Such not always the case
   Let’s anticipate some of problems that might be encountered

Local vs. Remote Addresses and Data
- Not all tasks are going to
  Be using same memory space
  Be on the same machine
- Different addressing schemes and data formats
  Must be carefully considered
- Data items in programs are expressed as primitives
  Arrays, structs, classes
  Richer more complex structures built from these
- Information being exchanged is inherently flat or sequential
- Not all processors express data the same way
  Different endianess or different word sizes are common
To permit the exchange of information among computers
  Must ensure that data values are expressed
    In an agreed upon common (external) form
Part of establishing a connection may be
  Negotiating common language
  Elect to communicate in some native form

Repeated Task Execution
   Must consider
   Request for action or procedure invocation from a remote device
     May become corrupted and hence rejected

Possibility that the complete message may never be received
   Consequently the remote procedure may
     Never be executed
     Be partially executed
     Be completely executed

Any of the alternatives can potentially lead to serious safety problems
   Complete execution of requested remote procedure may seem innocuous
   Consider request of the form
     ‘decrease flow rate of an inhibitor’
'increase the temperature of a process'

If confirmation that task was completed
Not received or properly interpreted by sender

Additional requests may be issued

Repeated requests may create serious safety problems

When designing exchange must consider
• How to avoid such duplicate messages,
• How to handle missed acknowledgments, and
• How to handle both the success and failure of an operation

To anticipate and manage such situations
Contemporary distributed embedded applications incorporate
*At most once semantics* and *atomic transactions*

Node Failure, Link Failure, Message Loss
Distributed system is susceptible to failure modes not seen in local model
Can generally detect failure
Often not possible to distinguish between
Link failure
Node failure
Message loss
Once fault is detected appropriate action can be taken.

Local Procedures and Remote Procedures
With those preliminary considerations in mind
Let’s get to work

A key thought here is ensuring invisibility of the infrastructure

A Traditional (Local) Procedure Call
In traditional software application
Most commonly used means for encapsulating a set of software instructions
A procedure
*Procedure call* is executed on main processor
By writing
Name of the procedure
Followed by the associated parameters enclosed in parentheses

In base case when called
Flow of control switches to procedure body
Execution at calling point
Blocked
Execution within procedure body commences
Continues until exit statement encountered
Flow of control switches back to calling point

A Remote Procedure Call
Distributed Programs
Can be viewed as set of software components
Running on number of (remote) computers in network

In client-server paradigm
Users interact with application programs
Which may be clients of any servers on network

Remote procedure calls
Similar to yet different from familiar local procedure calls

Even though procedure resides in remote address space
Would still like to be able to use semantics similar to local call

Support for remote call generally includes
An interface language processor,
A binding service
A communication driver

Invocation of remote procedure known as a remote procedure call - RPC
One may also see terminology remote procedure invocation - RPI used

Invocation most commonly based upon request-reply protocol
➢ Client invokes a service by sending request messages to server
➢ Server performs requested service and sends a reply back to client
   Generally client waits for reply before proceeding
   Analogous to a local call

Calling a Remote Procedure - RPC Semantics

Remote procedure call paradigm
Integrates client-server with conventional procedure call
Clients now interact with local and remote procedures
In seamless way

When a local procedure is called
Parameters and any return value
Usually passed into and returned from procedure via a stack
Following the call
  Calling procedure blocks waiting for the return

Such an approach is not possible with a remote procedure
  Still desirable that the remote call appear as if it had been local

Remote Procedure Call Features
  As noted goal of remote procedure call:
  To maintain semantics of conventional call

Presteps
  First step in creating a local call illusion
  Remote procedure specifies input and output parameters
  Write stubs for the procedure
    • They have the same public interface as the full procedure
      Same procedure name
      Return type
      Signature
    • The public interface masks the behind the scenes magic

These are then placed on the client and server

The call
  When the server process is ready
  It will execute a blocking receive
  Waiting for the call

When the client performs the call
  • Input parameters are passed to the server
    As values to arguments in a request message
    Equivalent to pass by value parameters
    In conventional procedures

Pass by reference parameters
  More difficult to implement
  Require additional information indicating use as
  Input output or both
  Executed in different environment from caller
  Cannot access
    Global or other variables in calling environment
  Passing memory addresses or equivalent
    Meaningless
  Args cannot include data structures or pointers
    To memory locations

  • Output parameters
Returned to client in reply message
Replace values of corresponding variables
In calling environment

To model local call semantics following the call
The client blocks and awaits the return.

Passing and Returning Parameters
Following block diagram illustrates the structure and flow of the operation

When the client invokes a procedure on a remote server
Call is intercepted by the client stub

Stub contains routines to put
✓ Parameters
✓ Return values
✓ Any other data to be interchanged

Into a format compatible with the network and with the remote system

Such a conversion is called marshalling

The stub then
Builds a message containing that information
Forwards that request message to the transport driver

The message is sent out over the network to the server node
The server node reverses the process
Converting from a network compatible format to format of the remote device
Called unmarshalling

Response from the server following the execution of the requested procedure
Follows the same procedure to return the reply back to the client

The remote procedure and associated operations
Have the same structure as a local version

Marshalling
   Begins with a collection of data types
   Flattens or serializes those structures
   Into a sequence of basic data items
   Data items converted into a form
   Suitable for transmission in an outgoing message
   That is they are translated into an external representation

Marshalling can be “done by hand”
   In sending program
   Generated automatically
      From a formal specification of data items to be transmitted

Conversion “done by hand”
   Means that the sending program explicitly converts each of the data items
      From their internal (local) representation
      To the agreed upon external (remote) representation

Unmarshalling
   Process of reassembling the data to its equivalent form
   May not be an identical form
   On arrival at remote node or at local node when a response is returned

Controlling the Transfer
Single message transfer is
Supported by send and receive operations

To communicate
   One process sends a message to a destination
   The destination process receives the message

Earlier, communication between the sending and receiving processes
   Was identified as synchronous or asynchronous
      With respect to a common clock

In current context these words have slightly different meanings.
**Synchronous exchange**
A accomplished by requiring that when a send is issued
  - Client process **blocks** and waits until corresponding receive is issued
    - Before sending out the next request

Similarly, the receiving process will block and await an incoming message

Such an exchange is called **Idle RQ or Stop-and-Wait.**

**Asynchronous exchange**
Significant improvements in performance can be gained
  - If the constraints on receiving the reply are relaxed

Under such a scheme following a send
  - Client process does **not block** - it is allowed to proceed
    - As soon as the outgoing message is copied into a local buffer

If client does not block
  - Client and the server are working in parallel
  - Client can send consecutive messages without waiting for a reply

Similarly the server may queue up several reply messages
  - While working on the next call

Such a scheme is similar to pipelining techniques
  - Used to improve performance of
    - CPU instruction cycle
    - Certain kinds of computations

The **non-blocking** scheme can be used if
  - The client tasks are computationally intense
  - The client is working on a task that requires coordination
    - With a number of servers

Such an exchange is called **Continuous RQ**

In **blocking** case
  - Client can proceed with the next computation
    - While the server is working with the results of the first

In **non-blocking** case
  - Client can send all the requests off
    - Then collect the replies as they come in

Such might be the case when
Client is in a master control console in an automated factory
Servers are at the various assembly lines

Client can
Send out status request messages to each of the distributed lines
Then collect the results

Little is gained by forcing the client to wait for each reply before sending the next

Places - Message Source and Destination
In the client-server model
There are a number of possible destinations for a message
- Process or group of processes
- Port or group of ports
- Socket
- Object

Consequently one of the arguments of a send operation
An identifier indicating the destination of the message
Most operating systems use a process or port
A port is a message destination that has
Exactly one receiver
Potentially many senders

On occasion, a port is also known as a mailbox
It's important to note
A mailbox always has a message queue
A port may not

In Internet protocols
Destination address is specified as
A port number used by the process and
The Internet address of the computer
On which the receiving process is running

The problem that arises with such an approach
The service must always be run on the same machine
For the address to remain valid

Ideally one would like to have location transparency
To that end location independent identifiers are used
Such an identifier is mapped by network driver and router software
Into a lower level address to deliver the message
A approach naturally takes into account
Current location of the service
Enabling message destinations and services to be changed
Without having to inform the clients of the new locations

When working with a client-server implementation
Communication is always in the form of request-reply pairs

Normally the communication is *synchronous*
The client process blocks until the reply is returned

One can elect to implement an *asynchronous* scheme
When the client can afford to delay retrieval

The Protocol
There are two primary ways by which a remote procedure call capability can be incorporated into a design

One is to utilize a programming language
In which the mechanism is already built-in

A disadvantage of such an approach is that the RPC requirements can be dealt with by language constructs
Similar to the way exceptions are handled in C++

An alternative is to use a special purpose interface language
Advantage of this approach is that the design is not tied to a particular language or language environment

The client-server protocol
Often implemented as trio of messages
- DoOperation
- GetRequest
- SendReply

Communications costs are low
Only three system calls are required

Server reply message is interpreted as an acknowledgement

Exchange given in accompanying state diagram

**RPC Interface Definition**
When writing a program in C or C++
Prototype for each of the functions in the design is specified

That information is used by the compiler
To ensure the proper binding of the procedure call to the procedure body
An RPC interface definition provides the same information in a list of procedures and associated signatures.

List identifies:
- The procedures
- The variables types
- The parameter types

For remote procedures, we identify three basic parameter types:

**Input only**
- Only permitted to pass information into a remote procedure

**Output only**
- Only send information from the server to the client
- Same parameter cannot be used to send information back to the server

**Input and Output**
- Parameter can be used
  - To send information from client to server
  - To return information from server to client

List enables the RPC system to identify which values to marshal into request and reply messages.

Also included in the interface definition:
- Similar information for procedures offered by the server visible to clients
- The service name is used by the clients and servers to refer each of the procedures.

**Node Failure, Link Failure, Message Loss Revisited**

As noted earlier, the requirements in today's embedded systems are placing high demands on safety and reliability.

An issue that is closely related to detecting and managing faults managing the delivery of messages.

When working in a distributed context, the problem of duplicate and lost messages is more acute than in a local context because there is greater opportunity for corruption.

**Node Failure and Loss**

As an aid to the early identification of a link or node failure, one can use a handshaking protocol monitored by a *watch dog timer*.

At periodic intervals, both sites send an *I-am-up* message.
If the watch dog timer expires
- Client is not be able to contact the server
- Time-out has occurred
  Using a request-reply-acknowledge protocol
Because a failure has occurred

Further action must be taken
The client can respond by
- Polling server or source of the expected I-am-up message
  If there is no response
  No additional information has been gained
- Trying second route
  If there is a response
    Known that the server is up and the link failed
  Else
    It is known
      The server is down
      Time-out is too short

If lengthening the time-out fails
Server failure can be assumed
Such reasoning presumes no double failure

Suppose that the detection protocol identifies a problem
When such exceptions occur
Client process must be able to
Report and manage them in a safe and robust way

- For software exceptions
  Some languages such as C++ and Java
  Provide constructs for handling them

  When no such means exist within the implementation language
  Procedure call can at minimum return an error code

- For hardware exceptions
  One can initiate a protocol to allow the system
  To reconfigure to continue operation

- If the direct link to the receiver has failed and been identified
  The information must be rebroadcast to all other sites in system
  Thereby allowing the routers to reconfigure

- If the client believes the site has failed
  Every other site must be notified
So they will not attempt to use failed node

When a failed site is repaired
   It must be reintegrated into system

Reintegration can be done by a handshaking procedure

Information to update routing tables also needs to be broadcast
   Queued messages may then be delivered to the site

Message Loss – Single Message
   The RPC semantics must specify what happens when
      Procedure call is repeated

In a distributed embedded application
   Always a finite probability that
      ➢ An initial procedure call
      ➢ A reply message
      ➢ A return value may be lost

Therefore it must be assumed that a procedure call may be repeated

Consider the following situation

✓ RPC executed to increase set point value on a boiler
✓ The call is received and executed
✓ Acknowledge is lost
✓ Sender times out on the response and resends

Leads to a potentially dangerous situation
   *The set point should not be raised a second time*

Exact implementation of such semantics
   Depends upon whether the receiver maintains state or not

When the server maintains state
   Client holds state information in some data structure
   Subsequent client calls to the server
      Build upon the stored information

   Server crash can potentially loose that information
      Unbeknownst to the client

When the server (and client) does not retain state information
   Each transaction stands alone.
Several of the more common RPC call semantics expressed as follows

**Maybe Call Semantics**
- No fault tolerance measures
  - If reply message not received after time-out
    - No retries
  - Uncertain if procedure executed
    - Procedure not executed if
      - Message lost
      - Server crashed
    - Procedure may be executed
      - Reply message lost
  - Such semantics unacceptable

**At Least Once Call Semantics**
- Retransmission of request messages without filtering of duplicates
  - Caller
    - Eventually given reply
    - Informed server presumed to have failed
  - Receiver
    - May receive and execute message more than once
    - If designed to be idempotent
  - Such semantics acceptable

**At Most Once Call Semantics**
- Filtering of duplicates retransmit replies
  - Without re-executing operation
  - RP designed to execute only once

Message Loss – Group Communication – Multicast
- Used for communication from single process to group of processes
  - Denoted Multicast

Useful tool for constructing distributed systems with the following characteristics

- Fault tolerance based upon replicated services
  - Client requests are multicast to members of group of servers
  - Each member performs identical operation
  - In event of failure clients still served
  - Several Multicast approaches used

**Atomic Multicast**
- Each server must receive and act on all requests
  - Denoted atomic Multicast
  - Ensures that all receivers are in same state after operation
  - If one does not receive then none receive
If server dies
Removed from group

An atomic transaction requires
Entire transaction be interpreted as
A single, indivisible unit of information

If the full request is not available
It is not acted upon

Reliable Multicast
Makes best effort to deliver to all members of group
Does not guarantee

- Locating objects in distributed systems
  Files within a distributed file system
  Multicast name query
  Only appropriate server responds
Once located
  File name can be cached to reduce further Multicast
Reliable Multicast acceptable
  State of system not changed
Re-Multicast acceptable

- Improved performance through replicated data
  Copies of data cached locally
  Speeds access
  Multicast used to update all copies when data changes

- Ordering
  In case 1 above
  Message ordering must be preserved
  Implemented through FIFO buffering on receivers

All receivers must execute commands in same sequence

Strongest ordering denoted
  Totally-ordered Multicast

Less restrictive is casual-ordering
  Assumes if two events occurred in same process
    They occurred in order observed
  Messages sent between processes
  Assume
    Event of sending message
    Occurred before event of receiving
Pipes, Streams, and Sockets

Now examine how one might make a logical connection to a (remote) process. Three alternatives are identified:

- Pipes
- Streams
- Sockets

Pipes

Earliest and most elementary type of interprocess communication mechanism.

They are simply an implementation of classic producer-consumer model.

Such a scheme allows two processes to communicate through a buffer of finite size.

That buffer is implemented as a shared FIFO data type.

Supports only one-way communication.

Created by a process using a system call. Analogous to the descriptors returned from a file access call.

System returns two pipe descriptors:
- One for reading
- One for writing

Following data flow diagram expresses such a relationship:

```
Process 0  Process 1
      ^       ^
      |       |
      v       v
    +------+
    | Pipe |
    +------+
```

The bidirectional flow illustrated in the figure:
- Made up of two unidirectional flows
- The interchanged data is stored in FIFO order.
- Using standard read and write operations.
- Communication is one-to-one.
- Pipe exists only as long as the processes do.

However, one process could:
- Perform a write operation.
-Terminate long before the consuming process.

When the processes are on separate compute engines:

Pipe is identified by the path to its location. Such pipes are referred to as named pipes.

Named pipes are restricted to one domain and a single file system.
Such a restriction is not particularly limiting.
For most distributed embedded applications.
Sockets

We have seen that a socket is means for logically connecting to a process
Communication identifier comprised of
Local port number
Internet address
Message destination specified as a socket address

With sockets
Interprocess communication operations
Based upon socket pairs
One belongs to each of pair of communicating processes
Implemented as exchange of information
Between socket on one process and socket on other
Messages queued at
Sending socket
Until network protocol is ready to transmit them
Receiving socket
Until receiving process ready to accept
Communication proceeds
Both or either direction
Between processes on
Same or different computers

Creation
Any process can create socket
Use for communication with another process
Socket call
Returns a descriptor by which socket referenced
In subsequent calls
Socket lasts until
Closed
Every process with the descriptor exits

Binding
Before pair of sockets can communicate
Recipient must bind
Socket descriptor to socket address
Sender must also
If reply required
System call used to execute binding
Requires
Socket descriptor
Reference to structure containing socket address
Once bound
Address cannot change
Stream Communication

To use stream protocol
Two processes first establish connection between their pair of sockets
Arrangement is asymmetrical
One is listening for connection
Other is asking for connection

Once connection established
Server
Forks a new process to communicate with client
Creates new socket
Pairs with client socket
Resumes listening in original process through original socket

Connection
Can be used to communicate in either or both directions
Data is read immediately in order transmitted
Continues until connection closed

The following pseudo code fragments
Illustrate setting up a client and server socket pair

```plaintext
clientSocket = socket(aDomain, aType, aProtocol);
code
bind(clientSocket, clientAddress);
code
sendto(clientSocket, myMessage, serverAddress);
code

serverSocket = socket(aDomain, aType, aProtocol);
code
bind(serverSocket, serverAddress);
code
recvFrom(serverSocket, myBuffer, clientAddress);
code
```

The `domain` specifies the communication domain
Which identifies the protocol family that will be used

The `type` identifies the semantics of the communication

The `protocol` stipulates the protocol that will be used for the communication

Today sockets are probably the most widely used interface
For message based interprocess communication.

Stream Communication
The notion of streams
Developed by Dennis Ritchie of Bell Labs in the early 1980’s

Pipes or the sockets provide an interface to a connection between two processes

A stream forms a connection between a device and an application process
  More specifically, the device driver

Communication is
  Byte-oriented
  Full duplex
  Characterized by
    A stream head
      Which interfaces to the application process
    The stream end or driver end
      Which connects to the driver software,
    Zero or more stream modules or processing modules

The stream modules provide
  Means for implementing and dynamically configuring device drivers

A module has
  An associated read queue for input
  A write queue for output

The modules
  Are pushed into the stream
    Where by the queues are connected together
      Much like a linked list in FIFO order
    To form a data flow between the stream head and the driver end

To use stream protocol
  Two processes first establish connection between their pair of sockets
  Arrangement is asymmetrical
    One is listening for connection
    Other is asking for connection

Once connection established
  Server
    Forks a new process to communicate with client
    Creates new socket
    Pairs with client socket
    Resumes listening in original process through original socket
  Connection
    Can be used to communicate in either or both directions
    Data is read immediately in order transmitted
    Continues until connection closed