CSci 5105

Introduction to Distributed Systems

Communication: MPI, MoM
MPI

• Message Passing Library Interface Standard

• For parallel computers, clusters, and heterogeneous networks

• Communication modes: standard, synchronous, buffered, and ready

• Designed to permit the development of portable parallel software libraries
# MPI Communication API

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_send</td>
<td>Send a message and wait until copied to local or remote buffer</td>
</tr>
<tr>
<td>MPI_ssend</td>
<td>Send a message and wait until receipt starts</td>
</tr>
<tr>
<td>MPI_sendrecv</td>
<td>Send a message and wait for reply</td>
</tr>
<tr>
<td>MPI_isend</td>
<td>Pass reference to outgoing message, and continue</td>
</tr>
<tr>
<td>MPI_issend</td>
<td>Pass reference to outgoing message, and wait until receipt starts</td>
</tr>
<tr>
<td>MPI_recv</td>
<td>Receive a message; block if there is none</td>
</tr>
<tr>
<td>MPI_irecv</td>
<td>Check if there is an incoming message, but do not block</td>
</tr>
</tbody>
</table>
Example: Startup

```c
#include <mpi.h>
#include <stdio.h>
int main(int argc, char *argv[])
{
    int rank, size;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    printf("I am %d of %d
", rank, size);
    MPI_Finalize();
    return 0;
}
```
MPI Abstractions

- Group: is the set of processes that communicate with one another
- Communicator: each communicator is associated with a group and a context
MPI Simple?

• Despite over 100 options ...
• Many parallel programs can be written using just these six functions
  – MPI_INIT
  – MPI_FINALIZE
  – MPI_COMM_SIZE
  – MPI_COMM_RANK
  – MPI_SEND
  – MPI_RECV

#include <stdio.h>
#include <mpi.h>

// No error checking
int main(int argc, char *argv[]) {

    const int tag = 42; // Message tag
    int id, ntasks, source_id, dest_id, st, i;
    MPI_Status status;
    int msg[2]; // Message array

    MPI_Init(&argc, &argv); // Initialize MPI
    MPI_Comm_size(MPI_COMM_WORLD, &ntasks); // Get # of tasks
    MPI_Comm_rank(MPI_COMM_WORLD, &id); // Get id

    if (ntasks < 2) {
        printf("You have to use at least 2 processors to run this program\n");
        MPI_Finalize(); // Quit if there is only one processor
        exit(0);
    }
}
if (id == 0) { /* Process 0 (the receiver) does this */
    for (i=1; i<ntasks; i++) {
        MPI_Recv(msg, 2, MPI_INT, MPI_ANY_SOURCE, tag,
                  MPI_COMM_WORLD, &status); // Receive a message
        source_id = status.MPI_SOURCE; // Get id of sender
    }
}
else { // Processes 1 to N-1 (the senders) do this
    msg[0] = id; // Put own identifier in the message
    msg[1] = ntasks; // and total number of processes
    dest_id = 0; // Destination address
    MPI_Send(msg, 2, MPI_INT, dest_id, tag, MPI_COMM_WORLD);
}

MPI_Finalize(); // Terminate MPI
exit(0);
return 0;
}
Motivation

• RPC and lower-level protocols assume sender/receivers run at the same time

• Suppose they are not?

• Need persistent communication
  – delay ~ of minutes is ok

• For users *and* applications
Scenarios

• Four possibilities
Message-Queuing Model

- Need a persistent queue
- Basic interface

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Put</td>
<td>Append a message to a specified queue</td>
</tr>
<tr>
<td>Get</td>
<td>Block until the specified queue is nonempty, and remove the first message</td>
</tr>
<tr>
<td>Poll</td>
<td>Check a specified queue for messages, and remove the first. Never block</td>
</tr>
<tr>
<td>Notify</td>
<td>Install a handler to be called when a message is put into the specified queue</td>
</tr>
</tbody>
</table>
Message-Queuing Model
Generalization

- Senders don’t care who they are sending to!
- Receivers don’t care who they are receiving from!
Architecture of a Message-Queuing System

• Series of queues with lookup services
• Automatic routing from source to destination queue
Relays

• Scalability
• Static routing
• Provide messaging features
  – logging
  – multicast
  – format transformation
Brokers

• Connect new applications
• Message formats are not compatible
• Application-gateway is needed to perform transformation for receiving clients
A network of brokers
IBM’s WebSphere

- **Message channels (MCA) connect message queues**
- **Specify specific behaviors**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transport type</td>
<td>Determines the transport protocol to be used</td>
</tr>
<tr>
<td>FIFO delivery</td>
<td>Indicates that messages are to be delivered in the order they are sent</td>
</tr>
<tr>
<td>Message length</td>
<td>Maximum length of a single message</td>
</tr>
<tr>
<td>Setup retry count</td>
<td>Specifies maximum number of retries to start up the remote MCA</td>
</tr>
<tr>
<td>Delivery retries</td>
<td>Maximum times MCA will try to put received message into queue</td>
</tr>
</tbody>
</table>
Programming in WebSphere
Use Cases?

• Application integration

• E-commerce
  – Past: an order went to a single application to process
  – Future: split order into multiple sub-orders and send to different vendors

• Different database query formats
Publish Subscribe

subscribe to a message with particular attributes
Example: Stock Quotes

- Stricter semantics on notification delivery
- Dealers for same stock should get same information
Finer-grain: Object-Based

Figure 5.10   Architecture for distributed event notification

events occur at an object of interest
notification: “event object”
observers propagate notifications
delivery semantics: real-time, reliable, ...
Programming Model: Tuple Space

```
take(<"String", "Scotland", String>)
write(<"Population", "Wales", 2900000>)

<"Capital", "Scotland", "Edinburgh">
<"Capital", "Wales", "Cardiff">
<"Capital", "N. Ireland", "Belfast">
<"Capital", "England", "London">
<"Population", "Scotland", 5168000>
<"Population", "UK", 61000000>
read(<"Population", String, Integer>)
take(<"String", "Scotland", Integer>)
```
Pub-Sub Architecture

Publish-subscribe architecture

- Matching

Event routing
- Flooding
- Filtering
- Rendezvous
- Informed gossip

Overlay networks
- Broker network
- Group multicast
- DHT
- Gossip

Network protocols
- TCP/IP
- IP mcast
- 802.11g
- MAC bcast
How does information propagate?

- **Flooding**
  - information is sent to all nodes
  - discard at destination

- **Filtering (network)**
  - brokers filter only if subscribers upstream

- **Filtering (source)**
  - flood subscriptions
  - filter at source
Propagation

• Advertisements
  – producers propagate advertisements towards subscribers

• Rendezvous
  – set of brokers responsible for specific portions of the event space
Next Time

Next topic: Streaming and Multicast

Read Chapters 4.4 - 4.5 TVS