ID1217 Concurrent Programming
Lecture 16

ROYAL INSTITUTE OF TECHNOLOGY

Tutorials:
MPI: Message Passing Interface.
Java Socket API

Vladimir Vlassov
KTH/ICT/EECS

Outline

- MPI: Message Passing Interface
  - The MPI library: functions, types;
  - MPI programming concepts: processes, communicators, etc.;
  - Basic MPI functions, blocking/non-blocking send/receive;
  - Collective operations: barriers, broadcast, gather/scatter, reduce, etc.;
  - Communicators.
- Java Socket API
  - TCP sockets: **`Socket`** (connecting socket) and **`ServerSocket`** (listening socket), streams
  - A client-server application using Java TCP socket API
  - Using UDP sockets: **`DatagramSocket`**, **`DatagramPacket`**, **`MulticastSocket`**

References

- MPI: Message Passing Interface
  - Ch 3 in "*An Introduction to Parallel Programming*", by P. Pacheco
  - Tutorial: Message Passing Interface (MPI) https://computing.llnl.gov/tutorials/mpi/
  - MPICH - A Portable Implementation of MPI http://www.mpich.org/
  - MPI Tutorials and Other documents http://www.mpich.org/documentation/guides/ http://www.mcs.anl.gov/research/projects/mpi/learning.html
  - Using the Hydra Process Manager http://wiki.mpich.org/mpich/index.php/Using_the_Hydra_Process_Manager
  - Chapter 8 "Message Passing Interface" in *Designing and Building Parallel Programs*, by Ian Foster http://www.mcs.anl.gov/~itf/dbpp/text/node94.html
- Java sockets
  - Trail: Custom Networking http://docs.oracle.com/javase/tutorial/networking/index.html
  - Java Networking http://docs.oracle.com/javase/8/docs/technotes/guides/net/index.html

MPI: Message Passing Interface

- MPI is a message-passing library specification for multiprocessor, clusters, and heterogeneous networks
  - Not a compiler specification, not a specific product.
  - Message-passing model and API
  - Designed
    - To permit the development of parallel software libraries
    - To provide access to advanced parallel hardware for end users, library writers, and tool developers.
  - MPI is a de facto standard for message passing systems
  - Language bindings: Fortran and C
- MPICH and LAM – the two most popular implementations of MPI today
  - MPICH is more generally usable on various platforms
  - LAM – for TCP/IP networks
- Other message-passing programming environment: PVM (Parallel Virtual Machine)

The MPI Library (API)

- The MPI library is large: about 130 functions
  - extensive functionality and flexibility
    - message passing (point-to-point, collective)
    - process groups and topologies
- The MPI library is "small" as using only 6 functions allow writing many programs
    - **`MPI_Init, MPI_Finalize, MPI_Comm_size, MPI_Comm_rank, MPI_Send, MPI_Recv`**
  - or another 6-function set:
    - **`MPI_Init, MPI_Finalize, MPI_Comm_size, MPI_Comm_rank, MPI_ Bcast, MPI_ Reduce`**
- All MPI functions, constants and data types have prefix **`MPI_`**

C Data Types Mapping in MPI

- (see `mpi.h`)

MPI datatype	C datatype
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

MPI Programming Concepts: Processes

- **A MPI process**
  - A smallest unit of computation, e.g. a Unix process.
  - Uniquely identified by its rank in a process group.
  - Processes communicate with tagged messages within process groups identified by communicators (communication contexts)
  - A proc can send messages either synchronously or asynchronously.
  - The proc is responsible for receiving messages directed to it.
  - The proc uses either blocking or non-blocking receive/send.
  - The proc can participate in a collective operation such as broadcast, gather, barrier, that involves all processes in a group.
- **A MPI application**
  - a static set of processes that interact to solve a problem.
  - `mpiexec -np 2 hello` – starts two `hello` processes.

Communicators. Process groups. Ranks

- **Communicator**
  - An opaque object that defines a process group and a communication context for the group. Delimits scope of communication.
    - Separate groups of processes working on subproblems each with specific communication context identified by a communicator.
- **Process group**
  - A virtual set of processes identified by a communicator.
    - Grouped for collective and point-to-point communications.
    - All communication (not just collective operations) takes place in groups.
    - Each process has a unique index (rank) in a given group.
- **Rank**
  - A unique identifier (index) of a process within a communicator (in a proc group)
  - Ranks are in the range 0, … , size-1 (where size is the size of the group)

Programming, Building and Running a MPI Application

- Install one of the MPI implementations, e.g. MPICH-2
- Develop your application – one program for all processes
  - The number of processes is specified at run time (could be 1)
  - Use the number of proc and proc ranks to determine process tasks
  - In C: `#include <mpi.h>`
- Compile:

  ```
  mpicc -o myprog myprog.c
  ```

  - For large projects, develop a Makefile
  - To compile C++: `mpicc –cc=g++ -o myprog myprog.cpp`
  - The option `-help` shows all options to `mpicc`.
- Run:

  ```
  mpiexec -np 2 myprog
  ```

  - The option `-help` shows all options to `mpiexec`.
  - Alternatively, you can call `mpirun` (which is a link to `mpiexec`)

Process Managers (1/2)

- **Process managers (e.g. Hydra, MPD)** are external distributed agents that spawn and manage parallel jobs.
  - A process manager communicates with MPICH processes using a predefined interface called as PMI (process management interface).
  - You can use any process manager from MPICH with any MPI application as long as they follow the same wire protocol.
  - There are three known implementations of the PMI wire protocol: "simple", "smpd" and "slurm".
  - By default, MPICH uses the "**simple**" PMI wire protocol, but can be configured to use "smpd" or "slurm" as well.
- MPICH provides several different process managers, e.g.
  - Hydra, MPD, Gforker and Remshell which follow the "simple" PMI wire protocol.

Process Managers (2/2)

- MPD has been the traditional default process manager for MPICH till the 1.2.x release series.
- Starting the 1.3.x series, **Hydra is the default process manager**.
- See "*Using the Hydra Process Manager*" at
  https://wiki.mpich.org/mpich/index.php/Using_the_Hydra_Process_Manager

Basic MPI Functions

- `int MPI_Init( int *argc, char **argv[] )`
  - Start MPI, enroll the process in the MPI application
- `int MPI_Finalize()`
  - Stop (exit) MPI
- `int MPI_Comm_size(MPI_Comm comm, int *size)`
  - Determine the number of processes in the group `comm`.
    - `comm` – communicator, e.g. `MPI_COMM_WORLD`
    - `size` – number of processes in group (returned)
- `int MPI_Comm_rank(MPI_Comm comm, int *rank)`
  - Determine the rank of the calling process in the group `comm`.
    - `comm` – communicator, e.g. `MPI_COMM_WORLD`
    - `rank` – the rank (returned) is a number between zero and `size-1`

Example: "Hello World"

```
#include "mpi.h"
#include <stdio.h>
int main( argc, argv )
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( "Hello world! I'm %d of %d\n", rank, size );
    MPI_Finalize();
    return 0;
}
```

Basic (Blocking) Send

```
int MPI_Send(void *buf, int count, MPI_datatype dt,
             int dest, int tag, MPI_Comm comm)
```

- Send a message with the given tag to the given destination process in the given communicator
  - `buf` send buffer
  - `count` number of items in buffer
  - `dt` data type of items
  - `dest` destination process rank
  - `tag` message tag
  - `comm` communicator
- Returns integer result code as for all MPI functions, normally `MPI_SUCCESS`
- *`datatype`* can be elementary, continues array of data types, stridden blocks of data types, indexed array of blocks of data types, general structure.

Basic (Blocking) Receive

```
int MPI_Recv( void *buf, int count, MPI_datatype dt,
              int source, int tag, MPI_Comm comm,
              MPI_Status *status)
```

– Receive a message with the given tag from the given source in the given communicator
  - `buf` receive buffer (loaded)
  - `count` max number of entries in buffer
  - `dt` data type of entries
  - `source` source process rank
  - `tag` message tag
  - `comm` communicator
  - `status` status (returned).

– "Wildcard" values are provided for tag (`MPI_ANY_TAG`) and source (`MPI_ANY_SOURCE`).

Inspecting Received Message

- If wildcard values are used for tag and/or sources, the received message can be inspected via a `MPI_Status` structure that has three components `MPI_SOURCE`, `MPI_TAG`, `MPI_ERROR`

```
MPI_Status status;
MPI_Recv( ..., &status );
int tag_received = status.MPI_TAG;
int rank_of_source = status.MPI_SOURCE;
MPI_Get_count( &status, datatype, &count );
```

  – `MPI_Get_count` is used to determine how much data of a particular type has been received.

Communication Modes of Blocking Send

- **Standard** blocking send (non-local) `MPI_Send`
  - Implementation defined buffering: If the message is buffered, the send may complete before a matching receive is invoked.
- **Buffered** blocking send (local) `MPI_Bsend`
  - Can be started whether or not a matching receive has been posted;
  - May complete before a matching receive is posted.
- **Synchronous** blocking send (non-local) `MPI_Ssend`
  - Can be started whether or not a matching receive has been posted;
  - Completes when a matching receive is posted and has started to receive.
- **Ready** blocking send (non-local) `MPI_Rsend`
  - May be started only if the matching receive is already posted;
  - Otherwise outcome is undefined.

ROYAL INSTITUTE OF TECHNOLOGY

Example 1: Exchange of Messages (Always Succeeds)

```
MPI_Comm_rank(comm, &rank);
if (rank == 0) {
    MPI_Send(sendbuf, count, MPI_REAL, 1, tag, comm);
    MPI_Recv(recvbuf, count, MPI_REAL, 1, tag, comm, &status);
}
if (rank == 1) {
    MPI_Recv(recvbuf, count, MPI_REAL, 0, tag, comm, &status);
    MPI_Send(sendbuf, count, MPI_REAL, 0, tag, comm);
}
```

- This program will succeed even if no buffer space for data is available.
- The standard send operation can be replaced, in this example, with asynchronous send.

Example 2: Exchange of Messages (Always Deadlocks)

```
MPI_Comm_rank(comm, &rank);
if (rank == 0) {
    MPI_Recv(recvbuf, count, MPI_REAL, 1, tag, comm, &status);
    MPI_Send(sendbuf, count, MPI_REAL, 1, tag, comm);
}
if (rank == 1) {
    MPI_Recv(recvbuf, count, MPI_REAL, 0, tag, comm, &status);
    MPI_Send(sendbuf, count, MPI_REAL, 0, tag, comm);
  }
```

– This program will always deadlock!
– The same holds for any other send mode.

Example 3: Exchange of Messages (Relies on Buffering)

```
MPI_Comm_rank(comm, &rank);
if (rank == 0)
  {
    MPI_Send(sendbuf, count, MPI_REAL, 1, tag, comm);
    MPI_Recv(recvbuf, count, MPI_REAL, 1, tag, comm, &status);
  }
if (rank == 1)
  {
    MPI_Send(sendbuf, count, MPI_REAL, 0, tag, comm);
    MPI_Recv(recvbuf, count, MPI_REAL, 0, tag, comm, &status);
  }
```

- For the program to complete, it is necessary that at least one of the two messages sent has been buffered.
- The program can succeed only if the communication system can buffer at least `count` words of data.

Non-Blocking Communication Operations

- **Non-blocking send** initiates sending:

```
int MPI_Isend(void* buf, int count, MPI_Datatype type,
              int dest, int tag, MPI_Comm comm,
              MPI_Request *request)
```

- **Non-blocking receive** initiates receiving:

```
int MPI_Irecv(void* buf, int count, MPI_Datatype type,
              int source, int tag, MPI_Comm comm,
              MPI_Request *request)
```

  – A request object is returned in `request` to identify the operation.

Non-Blocking Operations (cont'd)

- To query the status of communication or to wait for its completion:

```
int MPI_Test( MPI_Request *request, int *flag,
              MPI_Status *status)
```

  - Returns immediately with `flag = true` if the operation identified by `request` has completed, otherwise returns immediately with `flag = false`.

```
int MPI_Wait(MPI_Request *request, MPI_Status *status)
```

  - Returns when the operation identified by `request` completes.

Example: Non-Blocking Send/Receive

```
MPI_Comm_rank(comm, &rank);
if (rank = 0)
  {
    MPI_Isend(a, 10, MPI_REAL, 1, tag, comm, request);
    /**** do some computation to mask latency ****/
    MPI_Wait(request, &status);
  }
if (rank = 1)
  {
    MPI_Irecv(a, 10, MPI_REAL, 0, tag, comm, request);
    /**** do some computation to mask latency ****/
    MPI_Wait(request, &status);
  }
}
```

Probing for Pending Messages

`MPI_Iprobe(source, tag, comm, flag, status)`

– polls for pending messages

`MPI_Probe(source, tag, comm, status)`

– returns when a message is pending

- Non-blocking/blocking check for an incoming message <u>without receiving it</u>

ROYAL INSTITUTE OF TECHNOLOGY

Collective Operations

- A **collective operation** is executed by having all processes in the communicator call the same communication routine with matching arguments.
  - Several collective routines have a single originating or receiving process – the **root**.
  - Some arguments in the collective functions are specified as “significant only at root”, and are ignored for all participants except the root.

Collective Communication

ROYAL INSTITUTE OF TECHNOLOGY

- Collective synchronization (barrier):

```
int MPI_Barrier( MPI_Comm comm )
```

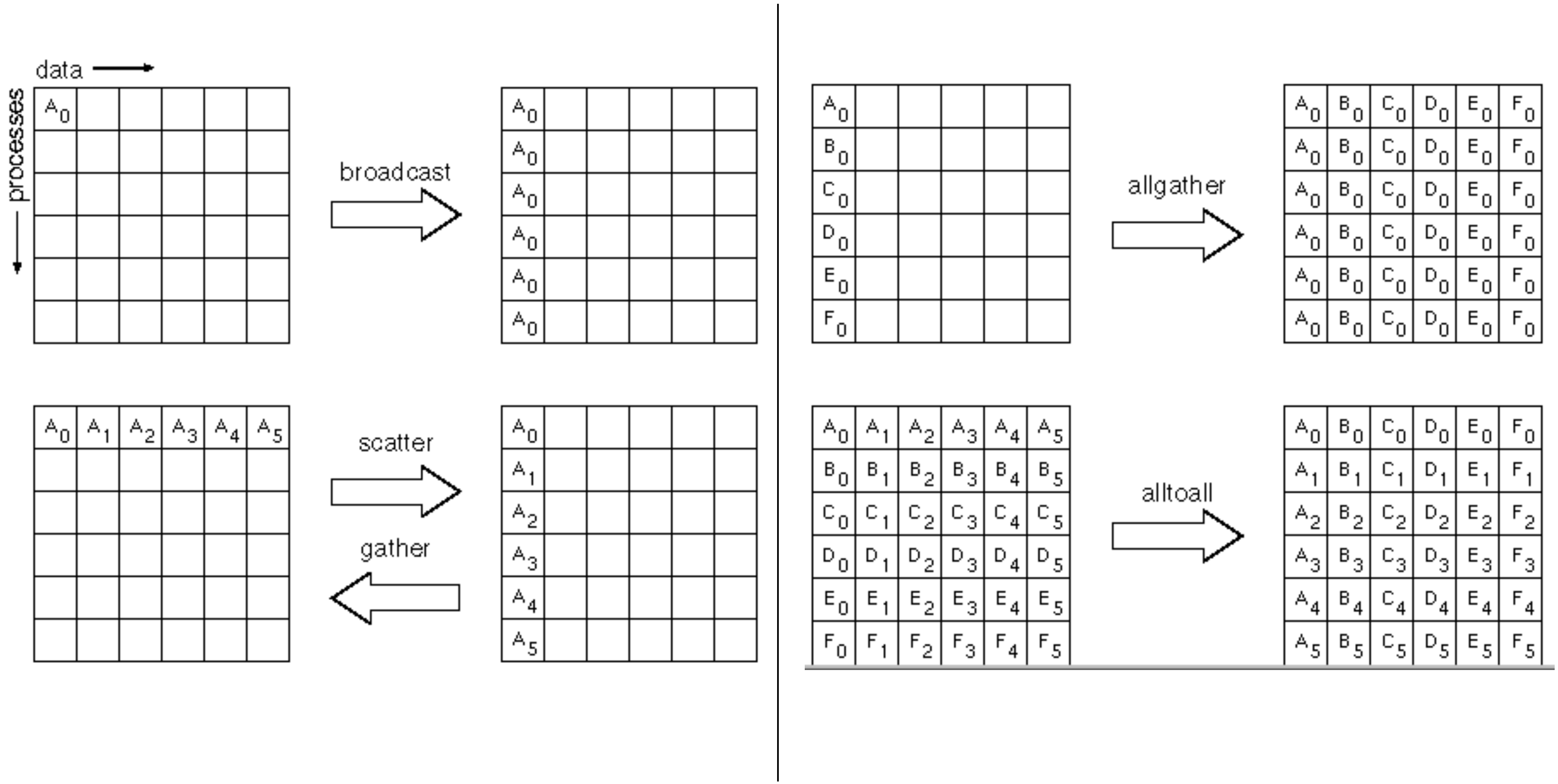
- Broadcast from **buf** of **root** to all processes:

```
int MPI_Bcast( void *buf, int count, MPI_datatype dt,
               int root, MPI_Comm comm )
```

- Collective data transfer:

```
int MPI_Gather( void *sendbuf, int sendcount,
                MPI_datatype sendtype, void *recvbuf,
                int recvcount, MPI_datatype recvtype,
                int root, MPI_Comm comm )
int MPI_Scatter( void *sendbuf, int sendcount,
                MPI_datatype sendtype, void *recvbuf,
                int recvcount, MPI_datatype recvtype,
                int root, MPI_Comm comm )
```

Collective Data Transfer Operations



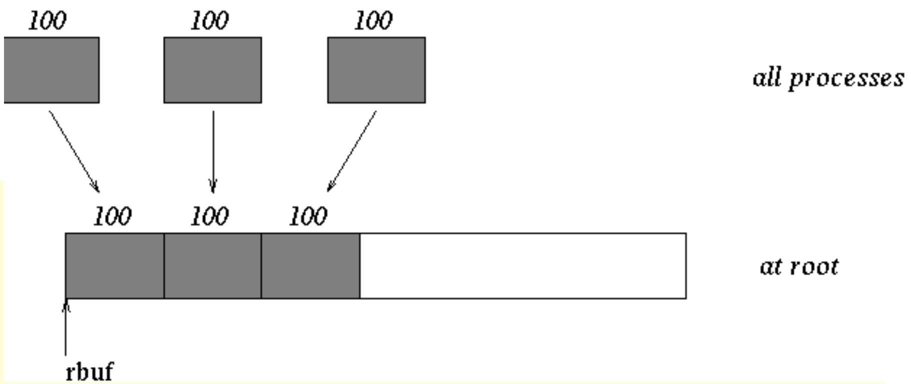
Example 1: Broadcast

- Broadcast 100 integers from process 0 (root) to every process in the group.

```
MPI_Comm comm;
    int array[100];
    int root=0;
    ...
    MPI_Bcast( array, 100, MPI_INT, root, comm);
}
```

Example 2: Gather

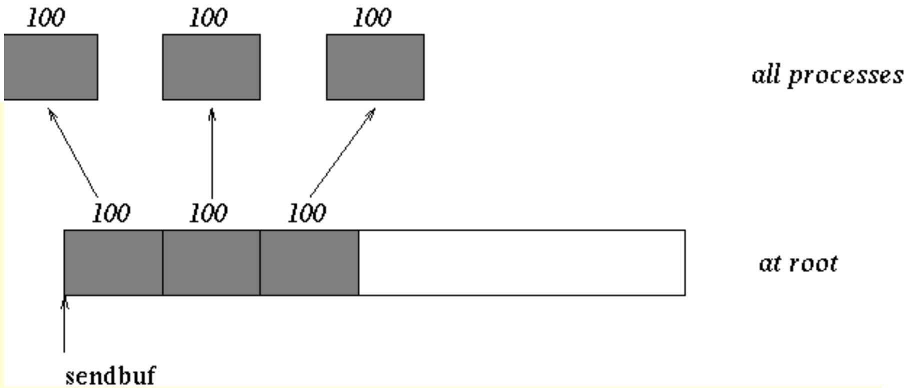
- Gather 100 integers from every process in group to root.




```
MPI_Comm comm;
int gsize, sendarray[100];
int root, myrank, *rbuf;
...
MPI_Comm_rank( comm, myrank);
MPI_Comm_size( comm, &gsize);
...
if ( myrank == root)
     rbuf = (int *)malloc(gsize*100*sizeof(int));
MPI_Gather( sendarray, 100, MPI_INT, rbuf, 100, MPI_INT, root, comm);
```

Example 3: Scatter

- The reverse of Example 2 (previous slide): Scatter sets of 100 integers from the root to each process in the group.




```
MPI_Comm comm;
int gsize,*sendbuf;
int root, rbuf[100];
...
MPI_Comm_rank( comm, myrank);
MPI_Comm_size( comm, &gsize);
if ( myrank == root) {
    sendbuf = (int *)malloc(gsize*100*sizeof(int));
    ... // fill the send buffer with data to be scattered
}
MPI_Scatter( sendbuf, 100, MPI_INT, rbuf, 100, MPI_INT, root, comm);
```

Global Reduction Operations

- Reduce data from send buffers of all participating processes into a receive buffer of the root proc using operation op

```
int MPI_Reduce( void *sendbuf, void *recvbuf,
                int count, MPI_datatype dt,
                MPI_Op op, int root, MPI_Comm comm )
```

- Reduce from send buffers of all participating processes into receive buffers of all the processes using operation op

```
int MPI_Allreduce( void * sendbuf, void * recvbuf,
                   int count, MPI_datatype dt,
                   MPI_Op op, MPI_Comm comm )
```

- Available operations (`MPI_Op op`) include:

  `MPI_MAX`, `MPI_MIN`, `MPI_SUM`, `MPI_LAND`, `MPI_BOR`, …

ROYAL INSTITUTE OF TECHNOLOGY

Timing Functions

- `double MPI_Wtime(void)`
  - Returns a floating-point number of seconds, representing elapsed wall-clock time since some time in the past. The time is “local” on the host.
- `double MPI_Wtick(void)`
  - Returns the resolution of `MPI_WTIME` in seconds, the number of seconds between successive clock ticks.
- Example of usage:

```
{
   double starttime, endtime;
   starttime = double MPI_Wtime();
    ....  stuff to be timed  ...
   endtime   = double MPI_Wtime();
   printf("That took %f seconds\n", endtime - starttime);
}
```

Modularity: Communicators

- A **communicator** identifies a process group and provides a context for all communication within the group
  - Communicator acts as an extra tag on messages
- The communicator `MPI_COMM_WORLD` identifies all running processes of the MPI application

Create/Destroy Communicators

- Create new communicator: same group, new context:

```
int MPI_Comm_dup(MPI_Comm comm, MPI_Comm *newcomm)
```

- Create new communicators based on colors, ordered by keys:

```
int MPI_Comm_split( MPI_Comm comm, int color, int key,
                    MPI_Comm *newcomm )
```

`color` identifies a group, `key` – rank in the group

- Create an inter-communicator from two intra-communicators

```
int MPI_Intercomm_create( MPI_Comm local_comm, int local_leader,
                          MPI_Comm peer_comm, int remoteleader,
                          int tag, MPI_Comm *inter_comm)
```

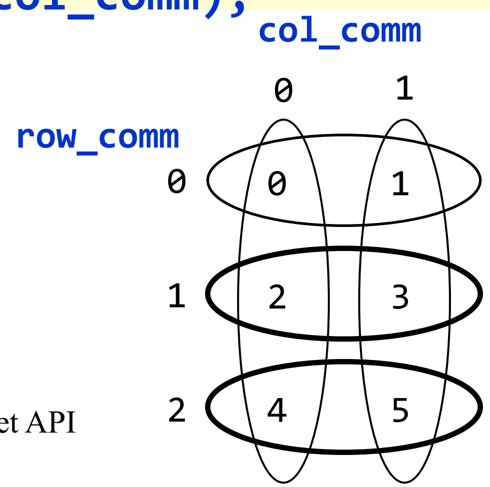
- Destroy a communicator

```
int MPI_Comm_free(MPI_Comm *comm)
```

Example MPI_Comm_split

```
// 2D topology with nrow rows and mcol (2) columns
// Split 3x2 grid into 2 communicators
// one (row_comm) corresponds to 3 rows;
// another (col_comm) – to 2 columns
irow = myID / mcol;        // logical row number
jcol = myID % mcol;        // logical column number
int row_comm, col_comm;
MPI_Comm_split(MPI_COMM_WORLD, irow, jcol, &row_comm);
MPI_Comm_split(MPI_COMM_WORLD, jcol, irow, &col_comm);
```

myID	0	1	2	3	4	5
irow	0	0	1	1	2	2
jcol	0	1	0	1	0	1



Communicators (cont'd)

- One use of communicators is for calling parallel library routines in different context:

```
MPI_Comm *newcomm;
...
MPI_Comm_dup(comm, newcomm);
transpose(newcomm, matrix);  /* call library function */
MPI_Comm_free(newcomm);
```

Example: Compute PI

```c
#include "mpi.h"
#include <math.h>
int main(int argc, char *argv[]) {
    int done = 0, n, myid, numprocs, i, rc;
    double PI25DT = 3.141592653589793238462643, mypi, pi, h, sum, x, a;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    while (!done) {
       if (myid == 0) {
          printf("Enter the number of intervals: (0 quits) ");
          scanf("%d",&n);
       }
       MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
       if (n == 0) break;
        h  = 1.0 / (double) n;
       sum = 0.0;
       for (i = myid + 1; i <= n; i += numprocs) {
          x = h * ((double)i - 0.5);
          sum += 4.0 / (1.0 + x*x);
       }
       mypi = h * sum;
      MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
                 MPI_COMM_WORLD);
      if (myid == 0)
        printf("pi is approximately %.16f, Error is %.16f\n",  pi, fabs(pi - PI25DT));
    }
  MPI_Finalize();
}
```

ROYAL INSTITUTE
OF TECHNOLOGY

Tutorial: The Java Socket API

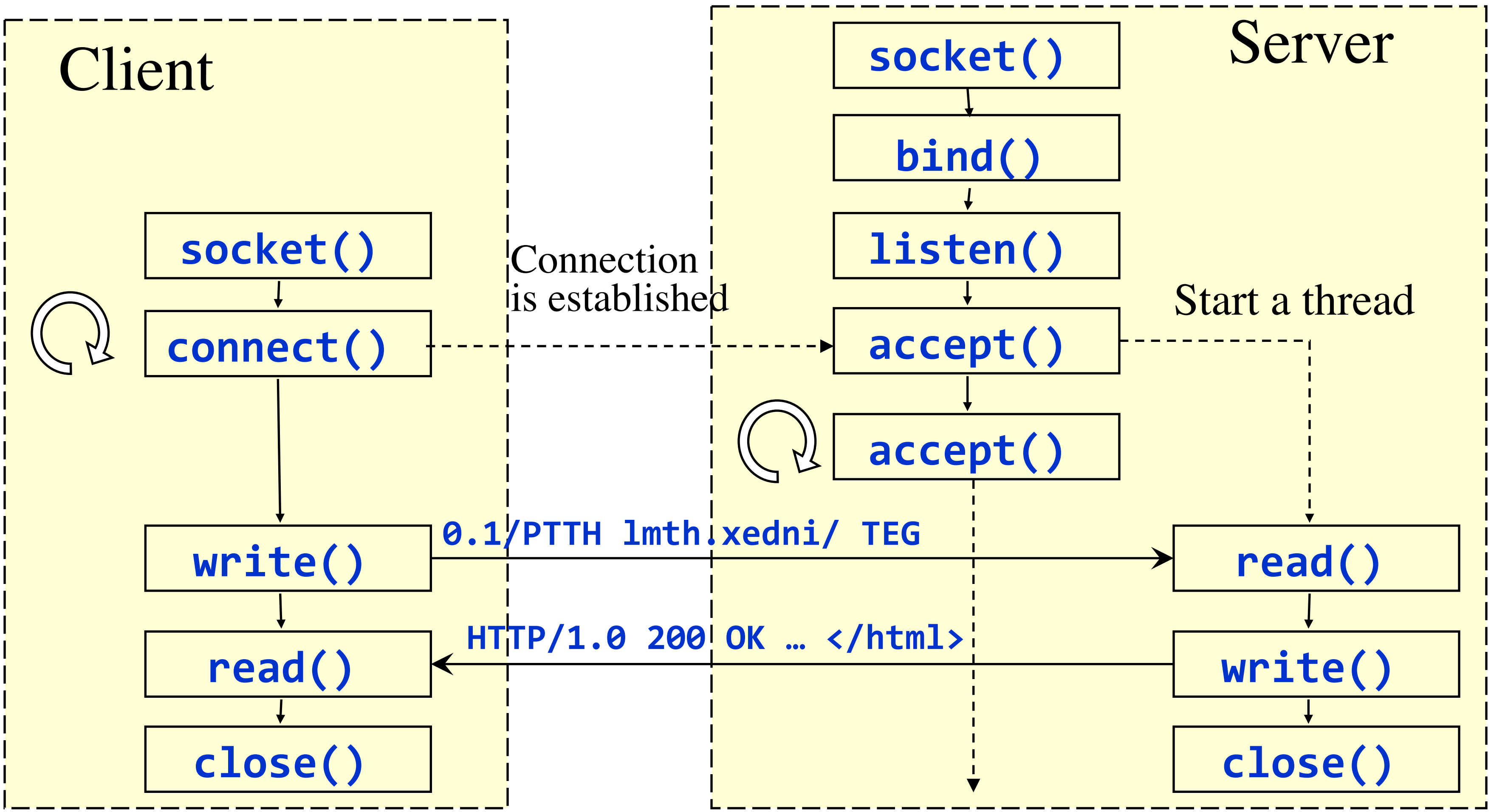
Sockets

- **Socket** is an end-point of a virtual network connection between processes
- Major parameters of a socket:
  - A socket address: a local IP address and a local port number, that the socket is bound to
  - A transport protocol used for communication via the socket
    - TCP socket - stream-based, connection-oriented
    - UDP socket - datagram-based, connectionless
- Sockets, a.k.a. Berkeley sockets, were introduced in 1981 as the Unix BSD 4.2 generic API for inter-process communication
    - Earlier, a part of the kernel (BSD Unix)
    - Now, a library (Solaris, MS-DOS, Windows, OS/2, MacOS)

Ports

- **A port** is an entry point to a process that resides on a host.
  - 65,535 logical ports with integer numbers 1 - 65,535
- A port can be allocated to a specific (well-known) service:
  - A server listens the port for incoming requests
  - A client connects to the port and requests the service
  - The server replies via the port.
- Ports with numbers 1-1023 are reserved for well-known services.
  - A list of services and allocated ports is stored in
    - /etc/services (Unix)
    - C:\Windows\services (Windows95)
    - C:\WINNT\system32\drivers\etc\services (WindowsNT, 2000)

The Berkeley Socket API for the Client-Server Architecture



Socket Classes in `java.net`***

- Two classes for TCP sockets:
  - `Socket` – Used to connect to another TCP socket specified by IP address and a port number.
    - When connected, provides two byte streams, input and output, used to communicate with the remote proc by reads and writes.
  - `ServerSocket` – Used to listen for connection requests, to accept a request and to create a socket connected to the requester.
- Two classes for UDP sockets:
  - `DatagramSocket` – Used for sending and/or receiving datagrams.
  - `MulticastSocket` is a subclass of DatagramSocket with capabilities for joining multicast groups on the Internet.

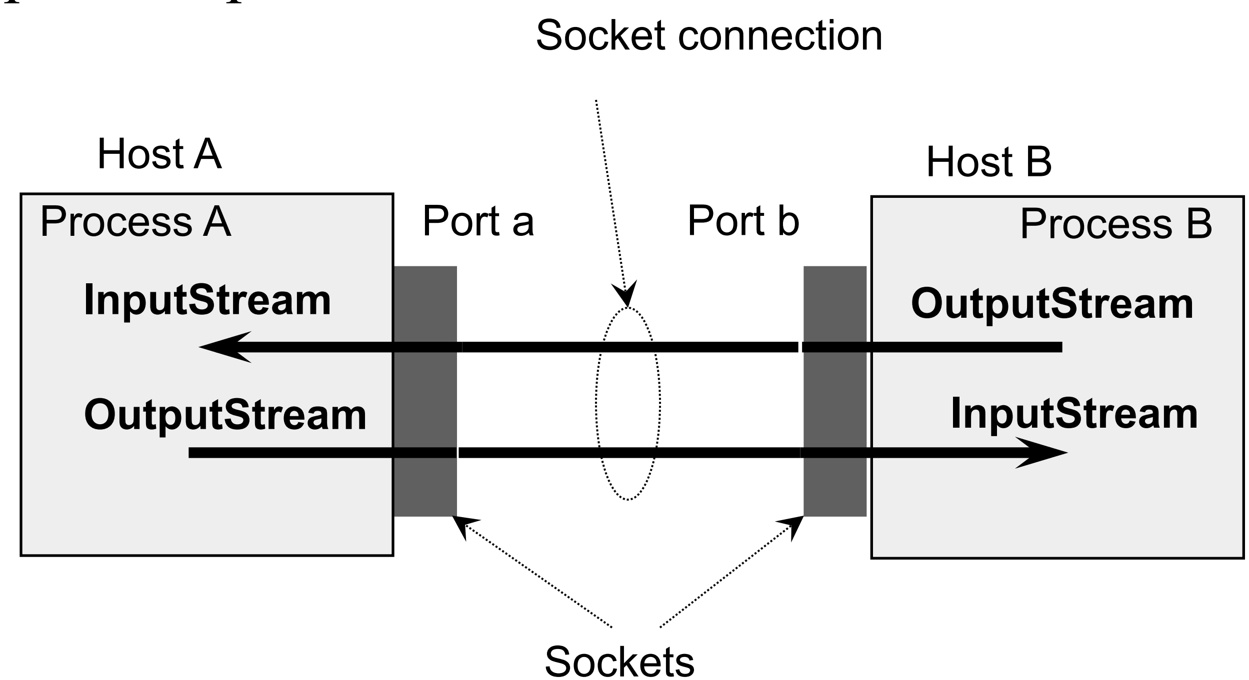
The `java.net.InetAddress` Class

- Represents an IP address of a node on the Internet.
- Does not have public constructors.
- Getting an IP address:

```
InetAddress ip1, ip2, localIP;
InetAddress[] ips;
try {
  // IP address of the local host
  localIP = InetAddress.getLocalHost();
  // IP address of a specified host
  ip1 = InetAddress.getByName("it-gw.it.kth.se");
       ip2 = InetAddress.getByName("130.237.214.1");
       ips = InetAddress.getAllByName("130.237.214.1");
} catch ( UnknownHostException e ) {
       e.print.StackTrace();
};
```

TCP Sockets. The `java.net.Socket` Class

- Implements a connecting ("client") TCP socket that provides connection (input and output byte streams) to a specified host on a specified port.



Socket Constructors

Socket(…)

```
(String remoteHost, int remotePort)

(InetAddress remoteAddr, int remotePort)

(String remoteHost, int remotePort,
 InetAddress localAddr, int localPort)

(InetAddress remoteAddr, int remotePort,
 InetAddress localAddr, int localPort)

( )

(SocketImpl socketImplementation)
```

Socket's Attributes

- **`setSoLinger(boolean, int)`**
  - Enable/disable SO_LINGER with the specified linger time (linger on close if data are present).
    - Note: Use **`netstat`** utility to check open connection.
- **`setSoTimeout(int)`**
  - Enable/disable SO_TIMEOUT with the specified time-out, in milliseconds, for blocking calls, e.g. read, connect.
- **`setTcpNoDelay(boolean)`**
  - Enable/disable TCP_NODELAY (disable/enable Nagle's algorithm).

Steps in Communicating via a TCP Socket

- Create a `Socket` object connected to a specified host on a specified port;
- Set socket's attributes, if needed;
- Get input and output streams of the socket connection.
- Communicate via the input and output streams by reads and writes according to an application specific communication protocol.
- Close the socket connection.

Example: A Code Fragment from a HTTP Client

```
try {
  // create a socket, open connection
  Socket socket = new Socket(host, port);
  // set timeout to 10 sec
  socket.setSoTimeout(10000);
  // get output stream
  PrintWriter wr = new PrintWriter(socket.getOutputStream());
  // send (write) GET request
  wr.println("GET " + file + " HTTP/1.0");
  wr.println();
  // flush output stream
  wr.flush();
  // get input stream
  BufferedReader rd = new BufferedReader(new
   InputStreamReader( socket.getInputStream()));
  // receive (read) and print response
  String str;
  while ((str = rd.readLine()) != null) System.out.println(str);
  socket.close(); // close connection
} catch (IOException e) {
   e.printStackTrace();  // communication failure
}
```

TCP Sockets. The `java.net.ServerSocket` Class

- Implements a listening ("server") TCP socket bound to some known port (and known IP address), to be used for listening and accepting connections from clients.
- Constructors:
  - `ServerSocket(int port)`
    - Create a server socket on specified port. A port of 0 creates a socket on any free port.
  - `ServerSocket(int port, int backlog)`
    - backlog is the maximum allowed length of queue of pending connection requests.
  - `ServerSocket(int port, int backlog, InetAddress bindLocalAddress)`

Accepting Connections

- **`Socket clientSocket = serverSocket.accept();`**
  - Blocks the current thread until a client connects
  - Returns a Socket object that represents the accepted connection.
- For example:

```
// create a server socket bound to the port 8080
ServerSocket serverSocket = new ServerSocket(8080);
while (true) {
    try {
        // wait for a client connection request
        Socket socket = serverSocket.accept();
        // communicate with the client connected
        …
        // close connection to the client
        socket.close();
    } catch (SocketException e) { e.printStackTrace(); }
}
```

Handling Connections

- The server uses a Socket object to communicate with a client that connects to the server
  - The server can handle the client in the same thread that accepts the connection.
  - The connection should be closed when the service is done.
- Concurrency for scalability – to service several clients simultaneously.
  - The server can communicate with each client in a separate thread.
  - When a client connects:
    - Construct a thread to handle the connection passing the Socket object as a parameter to the constructor of the thread.
    - Start the thread.
  - The main thread continues waiting for the other connection requests.

Outline of A Multithreaded Server in Java

```
{  . . .
    ServerSocket serversocket = new ServerSocket(8080);
    while (true) { // thread to accept connections and to start handlers
       try {
           Socket socket = serversocket.accept();
           Handler handler =  new Handler(socket);
           handler.setPriority(handler.getPriority()+1);
           handler.start();
       } catch (SocketException e) { e.printStackTrace(); }
    }
    . . .
}
class Handler extends Thread {
   private Socket socket;
   Handler(Socket socket) throws IOException { // thread constructor
        this.socket = socket;
        ... }
   public void run() { // communicate with the client
        ...
   }
}
```

UDP Sockets

- The **java.net.DatagramSocket** class
  - represents a UDP socket for sending and receiving datagrams – objects of the **java.net.DatagramPacket** class
- Sending datagrams:

```
DatagramSocket ds = new DatagramSocket();
byte buf[] = new byte[256];
// fill buf with data to sent
…
// create a datagram
DatagramPacket dp = new DatagramPacket( buf, buf.length,
        InetAddress.getByName(“dest.host.com“), 4711 );
// send the datagram via the UDP socket
ds.send(dp);
```

UDP Sockets (cont'd)

- Receiving datagrams:

```
byte b[] = new byte[256];
DatagramPacket dp = new DatagramPacket(b, b.length);
/* Set timeout – the amount of time (in milliseconds) that
   receive() waits for datagram before throwing an
   InterruptedIOException. With the time out of 0,
   receive() never times out.
*/
ds.setSoTimeout(timeout);
ds.receive(dp); // receive a datagram
byte[] data = dp.getData(); // get data
InetAddress source = dp.getAddress(); // source
int port = dp.getPort(); // source port
```

ROYAL INSTITUTE OF TECHNOLOGY

IP Multicast

- **IP multicast** – communication with a multicast group identified by IP address of the D class: 224-239.x.x.x
  - ~80 of multicast addresses are permanently assigned by the IANA (Internet Assigned Number Authority).
- **A multicast group** – a set of hosts sharing the same multicast address of the class D.
  - A host has to join a multicast group in order to receive datagrams directed to the group – it asks a default router.
  - A host does not need to join the group in order to send to the group.
- **TTL (Time-To-Live)** – a special field in the header of a IP packet
  - specifies the number (0-255) of routers the packet can pass through

Multicast in Java

- `MulticastSocket` is a subclass of `DatagramSocket` that represents a UDP socket with capabilities for joining multicast groups on the Internet.
- Communicating within a multicast group
  - Construct a multicast socket;
  - Join a multicast group;
  - Send/receive data to/from the multicast group;
  - Leave the group.
- Note:
  - To send a datagram to a multicast group, the host does need to join the group;
  - a DatagramSocket object can be used for sending datagrams to a multicast group

Receiving from a Multicast Group

```
try {
   MulticastSocket ms = new MulticastSocket(9875);
    ms.joinGroup(InetAddress.getByName("224.2.127.254"));
    while (true) {
      ms.receive(dp);
      String s = new String(dp.getData(),0,0,dp.getLength());
      System.out.println(s);
   }
} catch (Exception se) {
   se.printStackTrace();
}
```

Sending to a Multicast Group

```
InetAddress iaddr = InetAddress.getByName("224.17.17.17");
DatagramPacket dp = new DatagramPacket(data, data.length,
                                 iaddr, port);
try {
     MulticastSocket ms = new MulticastSocket();
     ms.setTimeToLive(16); // set TTL = 16
     //ms.joinGroup(iaddr); // not needed for sending
     ms.send(dp);
     //ms.leaveGroup(iaddr); // not needed for sending
     ms.close();
} catch (SocketException se) {
     se.printStackTrace();
} catch (IOException ioe) {
     ioe.printStackTrace();
}
```