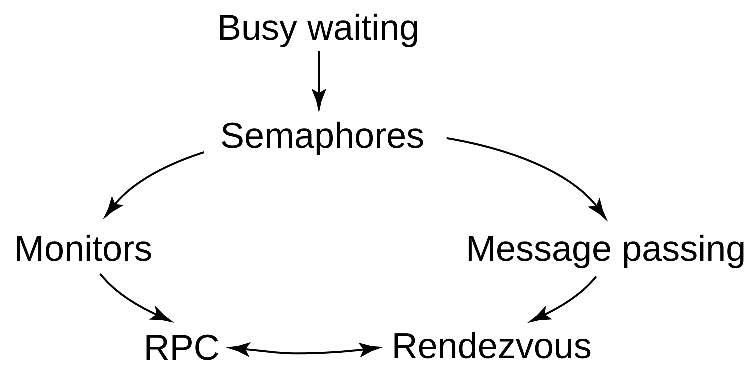
Distributed Programming

- Distributed memory architectures.
- Concurrent programs here are usually called *distributed* programs.
- No shared variables.
  - No mutual exclusion necessary!
- Communicate and synchronize with *channels*:
  - one-way or two-way
  - synchronous (blocking) or asynchronous (nonblocking)
- Four basic mechanisms:
  - Chapter 7:
    * asynchronous message passing
    * synchronous message passing
  - Chapter 8:
    * RPC (remote procedure call)
    * rendezvous
- Chapter 9 describes several paradigms for distributed programming:
  - managers/workers, hearbeat, pipeline, probe/echo, broadcast, token passing, decentralized servers



Relationships between programming mechanisms.

- Monitors combine implicit exclusion with explicit signalling
- Message passing adds data to semaphore
- RPC and Rendezvous combine procedural interface of monitors with implicit message passing

Message Passing

Andrews, Chapter 07

- Asynchronous message passing: channels are like semaphores.
- `send` and `receive` are like `V` and `P`
- The number of queued "messages" is the value of the semaphore.

```
chan input(char), output(char [MAXLINE]);
process Char_to_Line {
  char line[MAXLINE]; int i = 0;
  while (true) {
    receive input(line[i]);
    while (line[i] != CR and i < MAXLINE) {
      # line[0:i-1] contains the last i input characters
      i = i+1;
      receive input(line[i]);
    }
    line[i] = EOL;
    send output(line);
    i = 0;
  }
}
```

Figure 7.1 Filter process to assemble lines of characters.

```
chan in1(int), in2(int), out(int);
process Merge {
  int v1, v2;
  receive in1(v1);  # get first two input values
  receive in2(v2);
  # send smaller value to output channel and repeat
  while (v1 != EOS and v2 != EOS) {
    if (v1 <= v2)
      { send out(v1); receive in1(v1); }
    else   # (v2 < v1)
      { send out(v2); receive in2(v2); }
  }
  # consume the rest of the non-empty input channel
  if (v1 == EOS)
    while (v2 != EOS)
      { send out(v2); receive in2(v2); }
  else   # (v2 == EOS)
    while (v1 != EOS)
      { send out(v1); receive in1(v1); }
  # append a sentinel to the output channel
  send out(EOS);
}
```

Figure 7.2 A filter process that merges two input streams.

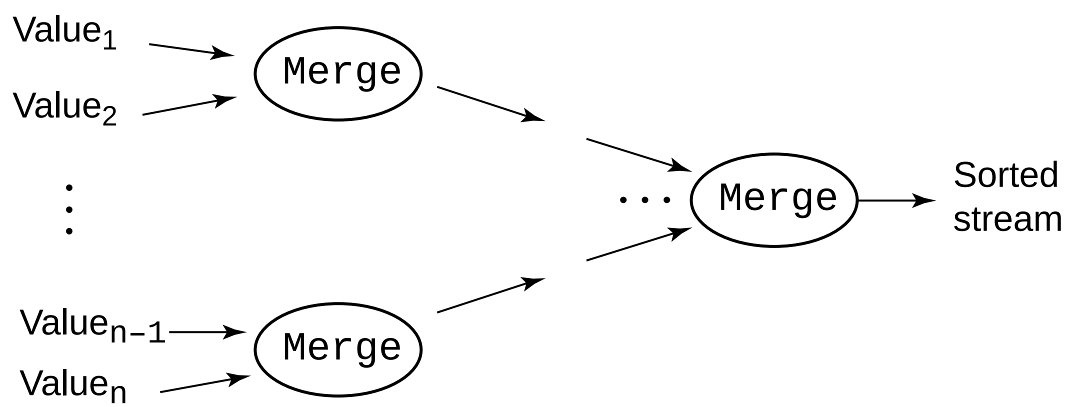


Figure 7.3 A sorting network of **`Merge`** processes.

```
chan request(int clientID, types of input values);
chan reply[n](types of results);
process Server {
  int clientID;
  declarations of other permanent variables;
  initialization code;
  while (true) {   ## loop invariant MI
    receive request(clientID, input values);
    code from body of operation op;
    send reply[clientID](results);
}
process Client[i = 0 to n-1] {
  send request(i, value arguments);     # "call" op
  receive reply[i](result arguments);   # wait for reply
}
```

Figure 7.4 Clients and server with one operation.

```
type op_kind = enum(op1, ..., opn);
type arg_type = union(arg1, ..., argn);
type result_type = union(res1, ..., resn);
chan request(int clientID, op_kind, arg_type);
chan reply[n](res_type);
process Server {
  int clientID; op_kind kind; arg_type args;
  res_type results; declarations of other variables;
  initialization code;
  while (true) {    ## loop invariant MI
    receive request(clientID, kind, args);
    if (kind == op1)
      { body of op1; }
    ...
    else if (kind == opn)
      { body of opn; }
    send reply[clientID](results);
  }
}
process Client[i = 0 to n-1] {
  arg_type myargs; result_type myresults;
  place value arguments in myargs;
  send request(i, opj, myargs);    # "call" opj
  receive reply[i](myresults);     # wait for reply
}
```

Figure 7.5 Clients and server with multiple operations.

```
monitor Resource_Allocator {
  int avail = MAXUNITS;
  set units = initial values;
  cond free;   # signaled when a process wants a unit
  procedure acquire(int &id) {
    if (avail == 0)
      wait(free);
    else
      avail = avail-1;
    remove(units, id);
  }
  procedure release(int id) {
    insert(units, id);
    if (empty(free))
      avail = avail+1;
    else
      signal(free);
  }
}
```

Figure 7.6 Resource allocation monitor.

```
type op_kind = enum(ACQUIRE, RELEASE);
chan request(int clientID, op_kind kind, int unitid);
chan reply[n](int unitID);
process Allocator {
  int avail = MAXUNITS; set units = initial values;
  queue pending;  # initially empty
  int clientID, unitID; op_kind kind;
  declarations of other local variables;
  while (true) {
    receive request(clientID, kind, unitID);
    if (kind == ACQUIRE) {
       if (avail > 0) {  # honor request now
          avail--; remove(units, unitID);
          send reply[clientID](unitID);
       } else   # remember request
          insert(pending, clientID);
    } else {   # kind == RELEASE
       if empty(pending) {  # return unitID to units
          avail++; insert(units, unitid);
       } else {  # allocate unitID to a waiting client
          remove(pending, clientID);
          send reply[clientID](unitID);
       }
    }
  }
}
process Client[i = 0 to n-1] {
  int unitID;
  send request(i, ACQUIRE, 0)    # "call" request
  receive reply[i](unitID);
  # use resource unitID, then release it
  send request(i, RELEASE, unitID);
   ...
}
```

Figure 7.7 Resource allocator and clients.

Monitor-Based Programs	*Message-Based Programs*
permanent variables	local server variables
procedure identifiers	**`request`** channel and operation kinds
procedure call	**`send request(); receive reply`**
monitor entry	**`receive request()`**
procedure return	**`send reply()`**
`wait` statement	save pending request
`signal` statement	retrieve and process pending request
procedure bodies	arms of case statement on operation kind

Table 7.1 Duality between monitors and message passing.

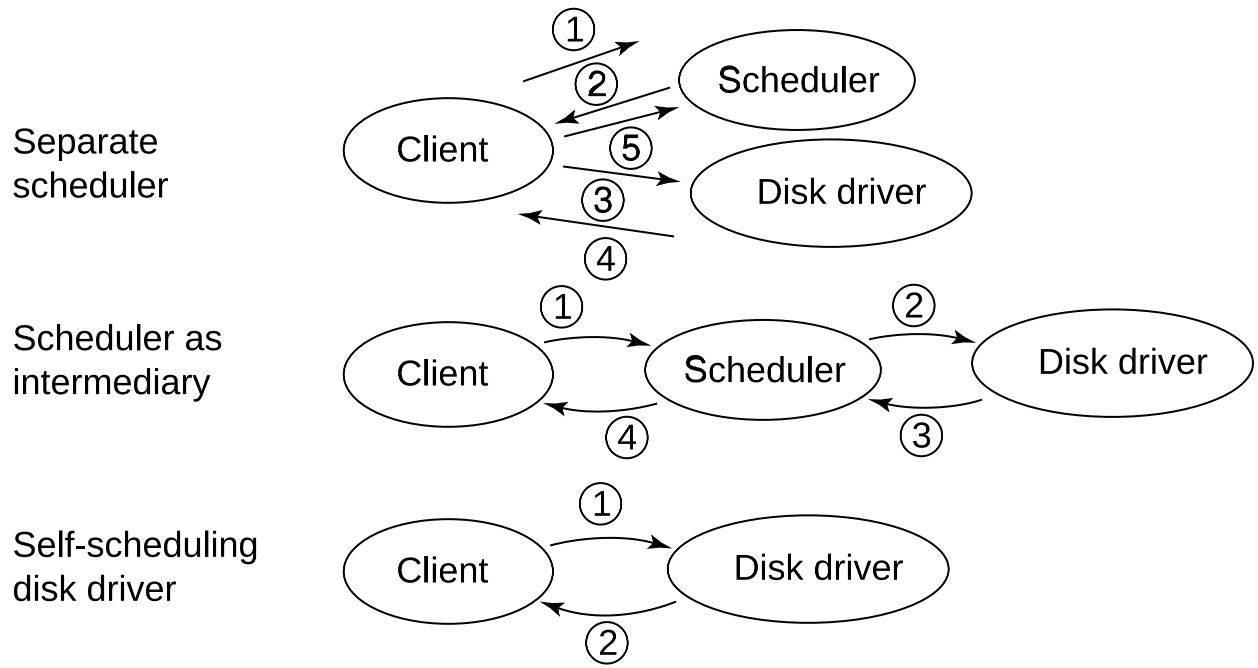


Figure 7.8 Disk scheduling structures with message passing.

```
chan request(int clientID, int cyl, types of other arguments);
chan reply[n](types of results);
process Disk_Driver {
  queue left, right;  # ordered queues of saved requests
  int clientID, cyl, headpos = 1, nsaved = 0;
  variables to hold other arguments in a request;
  while (true) {     ## loop invariant SST
    while (!empty(request) or nsaved == 0) {
      # wait for first request or receive another one
      receive request(clientID, cyl, ...);
      if (cyl <= headpos)
        insert(left, clientID, cyl, ...);
      else
        insert(right, clientID, cyl, ...);
      nsaved++;
    }
    # select best saved request from left or right
    if (size(left) == 0)
      remove(right, clientID, cyl, args);
    else if (size(right) == 0)
      remove(left, clientID, cyl, args);
    else
      remove request closest to headpos from left or right;
    headpos = cyl; nsaved--;
    access the disk;
    send reply[clientID](results);
  }
}
```

Figure 7.9 Self-scheduling disk driver.

```
type kind = enum(READ, WRITE, CLOSE);
chan open(string fname; int clientID);
chan access[n](int kind, types of other arguments);
chan open_reply[m](int serverID);  # server id or error
chan access_reply[m](types of results);  # data, error, ...

process File_Server[i = 0 to n-1] {
  string fname; int clientID;
  kind k; variables for other arguments;
  bool more = false;
  variables for local buffer, cache, etc.;
  while (true) {
    receive open(fname, clientID);
    open file fname; if successful then:
    send open_reply[clientID](i); more = true;
    while (more) {
      receive access[i](k, other arguments);
      if (k == READ)
        process read request;
      else if (k == WRITE)
        process write request;
      else   # k == CLOSE
        { close the file; more = false; }
      send access_reply[clientID](results);
    }
  }
}

process Client[j = 0 to m-1] {
  int serverID;  declarations of other variables;
  send open("foo", j);    # open file "foo"
  receive open_reply[j](serverID);  # get back server id
  # use file then close it by executing the following
  send access[serverID](access arguments);
  receive access_reply[j](results);
   ...
}
```

Figure 7.10 File servers and clients.

```
chan values(int), results[n](int smallest, int largest);
process P[0] {   # coordinator process
  int v;   # assume v has been initialized
  int new, smallest = v, largest = v;  # initial state
  # gather values and save the smallest and largest
  for [i = 1 to n-1] {
    receive values(new);
    if (new < smallest)
      smallest = new;
    if (new > largest)
      largest = new;
  }
  # send the results to the other processes
  for [i = 1 to n-1]
    send results[i](smallest, largest)
}
process P[i = 1 to n-1] {
  int v;   # assume v has been initialized
  int smallest, largest;
  send values(v);
  receive results[i](smallest, largest);
}
```

Figure 7.11 Exchanging values: centralized solution.

```
chan values[n](int);
process P[i = 0 to n-1] {
  int v;   # assume v has been initialized
  int new, smallest = v, largest = v;  # initial state
  # send my value to the other processes
  for [j = 0 to n-1 st j != i]
    send values[j](v);
  # gather values and save the smallest and largest
  for [j = 1 to n-1] {
    receive values[i](new);
    if (new < smallest)
      smallest = new;
    if (new > largest)
      largest = new;
  }
}
```

Figure 7.12 Exchanging values: symmetric solution.

```
chan values[n](int smallest, int largest);
process P[0] {  # initiates the exchanges
  int v;   # assume v has been initialized
  int smallest = v, largest = v;  # initial state
  # send v to next process, P[1]
  send values[1](smallest, largest);
  # get global smallest and largest from P[n-1] and
  #   pass them on to P[1]
  receive values[0](smallest, largest);
  send values[1](smallest, largest);
}
process P[i = 1 to n-1] {
  int v;   # assume v has been initialized
  int smallest, largest;
  # receive smallest and largest so far, then update
  #   them by comparing their values to v
  receive values[i](smallest, largest)
  if (v < smallest)
      smallest = v;
  if (v > largest)
      largest = v;
  # send the result to the next processes, then wait
  # to get the global result
  send values[(i+1) mod n](smallest, largest);
  receive values[i](smallest, largest);
}
```

Figure 7.13 Exchanging values using a circular ring.

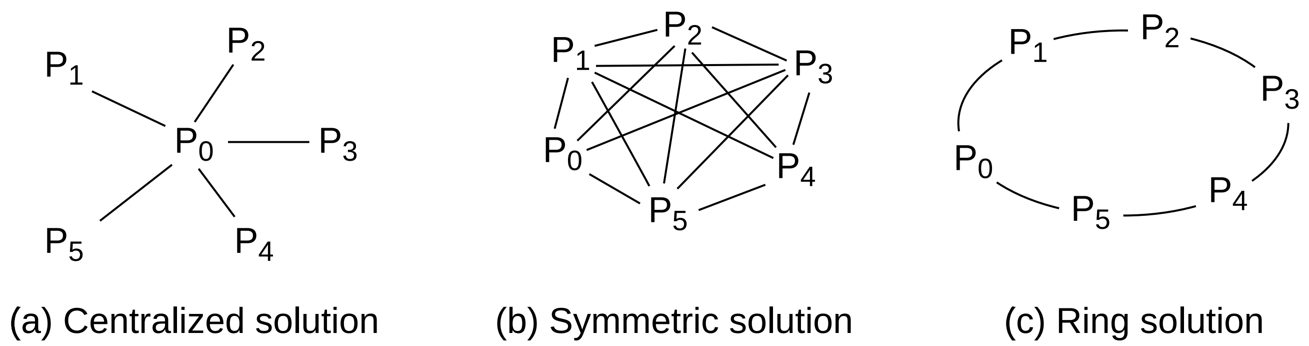


Figure 7.14 Communication structures of the three programs.

```
channel values(int);

process Producer {
  int data[n];
  for [i = 0 to n-1] {
    do some computation;
    synch_send values(data[i]);
  }
}

process Consumer {
  int results[n];
  for [i = 0 to n-1] {
    receive values(results[i]);
    do some computation;
  }
}
```

Producer/consumer example using synchronous message passing.

```
channel in1(int), in2(int);

process P1 {
  int value1 = 1, value2;
  synch_send in2(value1);
  receive in1(value2);
}

process P2 {
  int value1, value2 = 2;
  synch_send in1(value2);
  receive in2(value1);
}
```

Exchanging values with synchronous message passing.

```
process GCD {
  int id, x, y;
  do true ->
    Client[*]?args(id, x, y);  # input a "call"
    # repeat the following until x == y
    do x > y -> x = x - y;
    [] x < y -> y = y - x;
    od
    Client[id]!result(x);  # return the result
  od
}
```

```
 ... GCD!args(i,v1,v2); GCD?result(r); ...
```

Greatest common divisor process and interface in CSP.

```
process Copy {   # one character buffer
  char c;
  do West?c -> East!c; od
}
```

```
process Copy {   # two character buffer
  char c1, c2;
  West?c1;
  do West?c2 -> East!c1; c1 = c2;
  [] East!c1 -> West?c1;
  od
}
```

```
process Copy {   # ten character buffer
  char buffer[10];
  int front = 0, rear = 0, count = 0;
  do count < 10; West?buffer[rear] ->
        count = count+1; rear = (rear+1) mod 10;
  [] count > 0; East!buffer[front] ->
        count = count-1; front = (front+1) mod 10;
  od
}
```

Versions of copy processes in CSP.

```
process Allocator {
  int avail = MAXUNITS;
  set units = initial values;
  int index, unitid;
  do avail > 0; Client[*]?acquire(index) ->
        avail--; remove(units, unitid);
        Client[index]!reply(unitid);
  [] Client[*]?release(index, unitid) ->
        avail++; insert(units,unitid);
  od
}
```

Resource allocator in CSP.

```
process P1 {
  int value1 = 1, value2;
  if P2!value1 -> P2?value2;
  [] P2?value2 -> P2!value1;
  fi
}
process P2 {
  int value1, value2 = 2;
  if P1!value2 -> P1?value1;
  [] P1?value1 -> P1!value2;
  fi
}
```

Exchanging values in CSP.

```
process Sieve[1] {
  int p = 2;
  for [i = 3 to n by 2]
    Sieve[2]!i;   # pass odd numbers to Sieve[2]
}
process Sieve[i = 2 to L] {
  int p, next;
  Sieve[i-1]?p;                 # p is a prime
  do Sieve[i-1]?next ->         # receive next candidate
    if (next mod p) != 0 ->     # if it might be prime,
      Sieve[i+1]!next;          #   pass it on
    fi
  od
}
```

Figure 7.15 Sieve of Eratosthenes in CSP.

```
CHAN OF BYTE comm :
PAR
  WHILE TRUE     -- keyboard input process
    BYTE ch :
    SEQ
      keyboard ? ch
      comm ! ch
  WHILE TRUE     -- screen output process
    BYTE ch :
    SEQ
      comm ? ch
      display ! ch
```

Producer/consumer example in Occam.

```
PROC Copy(CHAN OF BYTE West, Ask, East)
  BYTE c1, c2, dummy :
  SEQ
    West ? c1
    WHILE TRUE
      ALT
        West ? c2    -- West has a byte
          SEQ
            East ! c1
            c1 := c2
        Ask ? dummy  -- East wants a byte
          SEQ
            East ! c1
            West ? c1
```

Copy process in Occam.

```
COPY1 = West?c:char -> East!c -> COPY1


COPY = West?c1:char -> COPY2(c1)

COPY2(c1) =   West?c2:char -> East!c1 -> COPY2(c2)
             []
              East!c1 -> West?c1:char -> COPY2(c1)


GCD = Input?id.x.y -> GCD(id, x, y)

GCD(id, x, y) = if (x = y) then
                  Output!id.x -> GCD
                else if (x > y) then
                  GCD(id, x-y, y)
                else
                  GCD(id, x, y-x)
```

Examples of Modern CSP.

```c
#include "linda.h"
#define LIMIT 1000     /* upper bound for limit */

void worker() {
  int primes[LIMIT] = {2,3};  /* table of primes */
  int numPrimes = 1, i, candidate, isprime;

  /* repeatedly get candidates and check them */
  while(true) {
    if (RDP("stop"))  /* check for termination */
      return;
    IN("candidate", ?candidate);    /* get candidate */
    OUT("candidate", candidate+2);  /* output next one */
    i = 0; isprime = 1;
    while (primes[i]*primes[i] <= candidate) {
      if (candidate%primes[i] == 0) { /* not prime */
        isprime = 0; break;
      }
      i++;
      if (i > numPrimes) {  /* need another prime */
        numPrimes++;
        RD("prime", numPrimes, ?primes[numPrimes]);
      }
    }
    /* tell manager the result */
    OUT("result", candidate, isprime);
  }
}

real_main(int argc, char *argv[]) {
  int primes[LIMIT] = {2,3};  /* my table of primes */
  int limit, numWorkers, i, isprime;
  int numPrimes = 2, value = 5;
  limit = atoi(argv[1]);  /* read command line */
  numWorkers = atoi(argv[2]);

  /* create workers and put first candidate in bag */
  for (i = 1; i <= numWorkers; i++)
    EVAL("worker", worker());
  OUT("candidate", value);

  /* get results from workers in increasing order */
  while (numPrimes < limit) {
    IN("result", value, ?isprime);
    if (isprime) {   /* put value in table and TS */
      primes[numPrimes] = value;
      OUT("prime", numPrimes, value);
      numPrimes++;
    }
    value = value + 2;
  }
  /* tell workers to quit, then print the primes */
  OUT("stop");
  for (i = 0; i < limit; i++)
    printf("%d\n", primes[i]);
```

```
}
```

Figure 7.16 Prime number generation in C-Linda.

```
#include <mpi.h>

main(int argc, char *argv[]) {
  int myid, otherid, size;
  int length = 1, tag = 1;
  int myvalue, othervalue;
  MPI_Status status;

  /* initialize MPI and get own id (rank) */
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &myid);

  if (myid == 0) {
    otherid = 1; myvalue = 14;
  } else {
    otherid = 0; myvalue = 25;
  }
  MPI_Send(&myvalue, length, MPI_INT, otherid,
           tag, MPI_COMM_WORLD);
  MPI_Recv(&othervalue, length, MPI_INT, MPI_ANY_SOURCE,
           tag, MPI_COMM_WORLD, &status);
  printf("process %d received a %d\n", myid, othervalue);

  MPI_Finalize();
}
```

Figure 7.17 MPI program to exchange values between two processes.

```
// Read a file and send it back to a client
import java.io.*; import java.net.*;

public class FileReaderServer {
 public static void main(String args[]) {
  try {
   // create server socket and
   // listen for connection on port 9999
   ServerSocket listen = new ServerSocket(9999);

   while (true) {
     System.out.println("waiting for connection");
     Socket socket = listen.accept();  // wait for client
     // create input and output streams to talk to client
     BufferedReader from_client =
       new BufferedReader(new InputStreamReader
         (socket.getInputStream()));
     PrintWriter to_client = new PrintWriter
         (socket.getOutputStream());

     // get filename from client and check if it exists
     String filename = from_client.readLine();
     File inputFile = new File(filename);
     if (!inputFile.exists()) {
       to_client.println("cannot open " + filename);
       to_client.close(); from_client.close();
       socket.close();
       continue;
     }

     // read lines from filename and send to the client
     System.out.println("reading from file " + filename);
     BufferedReader input =
       new BufferedReader(new FileReader(inputFile));
     String line;
     while ((line = input.readLine()) != null)
       to_client.println(line);
     to_client.close(); from_client.close();
     socket.close();
  }}
  catch (Exception e) // report any exceptions
    { System.err.println(e); }
}}
```

Figure 7.18 A file reader server in Java.

```java
// Get file from RemoteFileServer and print on stdout
import java.io.*; import java.net.*;

public class Client {
 public static void main(String[] args) {
  try {
   // read command-line arguments
   if (args.length != 2) {
     System.out.println("need 2 arguments");
     System.exit(1);
   }
   String host = args[0];
   String filename = args[1];

   // open socket, then input and output streams to it
   Socket socket = new Socket(host,9999);
   BufferedReader from_server =
     new BufferedReader(new InputStreamReader
       (socket.getInputStream()));
   PrintWriter to_server = new PrintWriter
       (socket.getOutputStream());

   // send filename to server, then read and print lines
   // until the server closes the connection
   to_server.println(filename); to_server.flush();
   String line;
   while ((line = from_server.readLine()) != null) {
     System.out.println(line);
   }
  }
  catch (Exception e)    // report any exceptions
    { System.err.println(e); }
}}
```

Figure 7.19 A file reader client in Java.