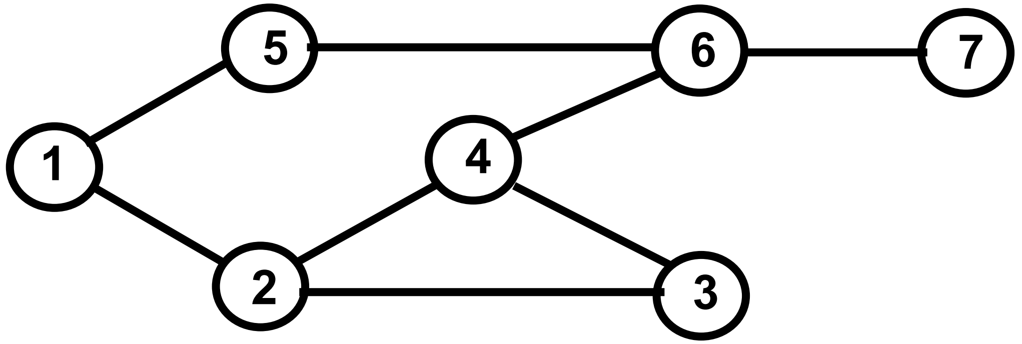
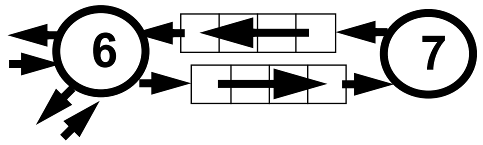
8. Messages in Distributed Systems
Distributed processes: Broadcast in a net of processors

Net: bi-directional graph, connected, irregular structure;
  node: a process
  edge: a pair of links (channels) which connect two nodes in both directions

A node knows only its direct neighbours and the links to and from each neighbour:





Broadcast:
  A message is sent from an initiator node such that it reaches every node in the net.
  Finally all channels have to be empty.

Problems:

- graph may have cycles
- nodes do not know the graph beyond their neighbours

Broadcast method

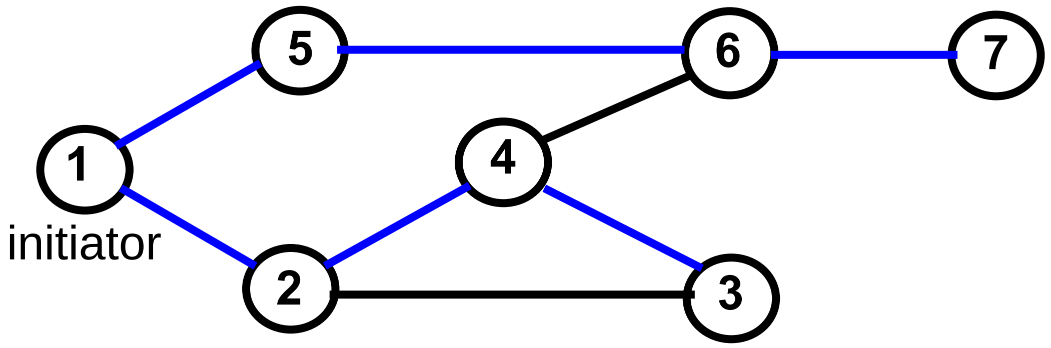
Method (for all nodes but the initiator node):

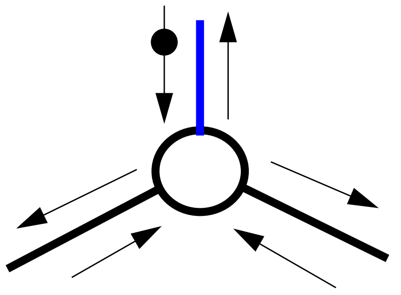
1. The node waits for a message on its incoming links.
2. After having **received the first message** it sends a **copy to all of its n neighbours** - including to the sender of the first message
3. The node then receives **n-1 redundant messages** from the remaining neighbours

All nodes are finally reached because of (2).

All channels are finally empty because of (3).

The connection to the sender of the first message is considered to be an edge of a **spanning tree** of the graph. That information may be used to simplify subsequent broadcasts.





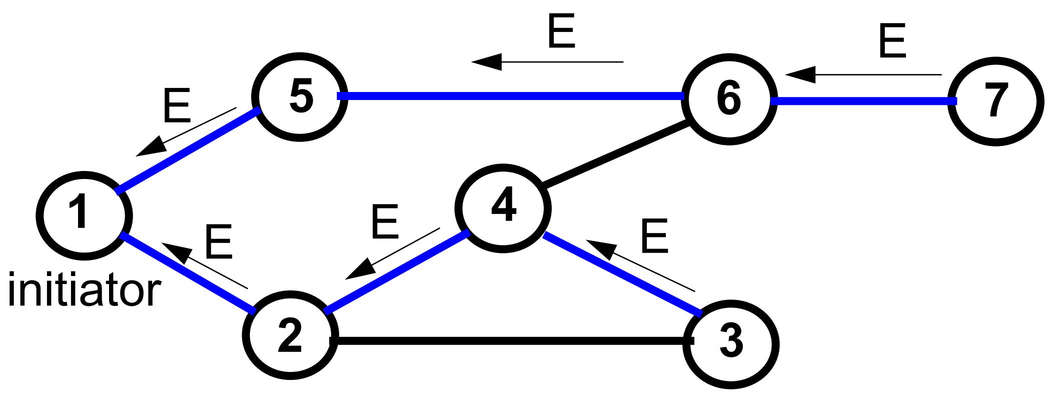
total number of messages: 2*|edges|

Probe and echo in a net

Task: An initiator requests combined **information from all nodes** in the graph (**probe**).
The information is **combined** on its way through the net (**echo**);
e. g. sum of certain values local to each node, topology of the graph, some global state.

Method (roughly):

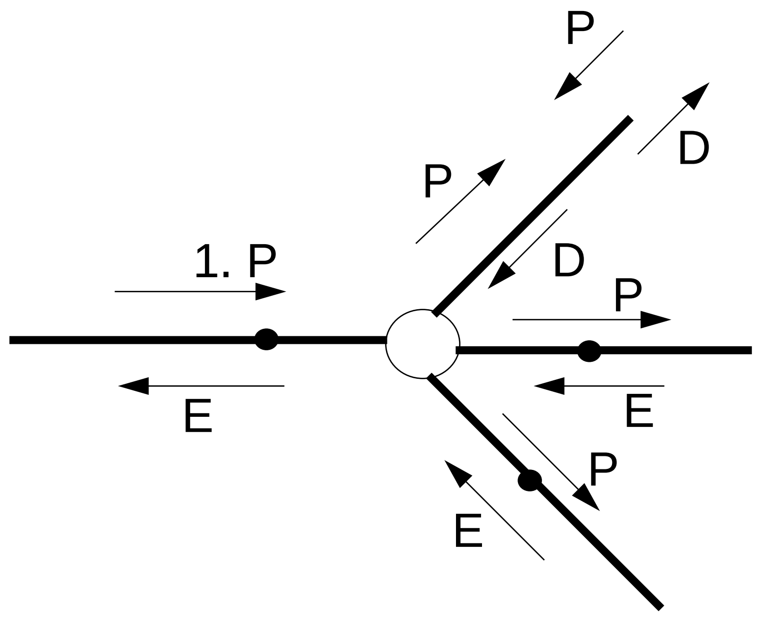
- distribute the probes like a broadcast,
- let the first reception determine a spanning tree,
- return the echoes on the spanning tree edges.



Probe and echo: detailed operations

Operations of each node (except the initiator):

- The node has **n neighbours** with an **incoming and outgoing link to each** of them.
- After having **received the first probe from neighbour s**, send a **probe to all neighbours except to s**, i. e. **n - 1 probes**.
- Each further **incoming probe** is replied with a **dummy** message.
- Wait until **n - 1 dummies and echoes** have arrived.
- Then combine the echoes and **send it to s**.

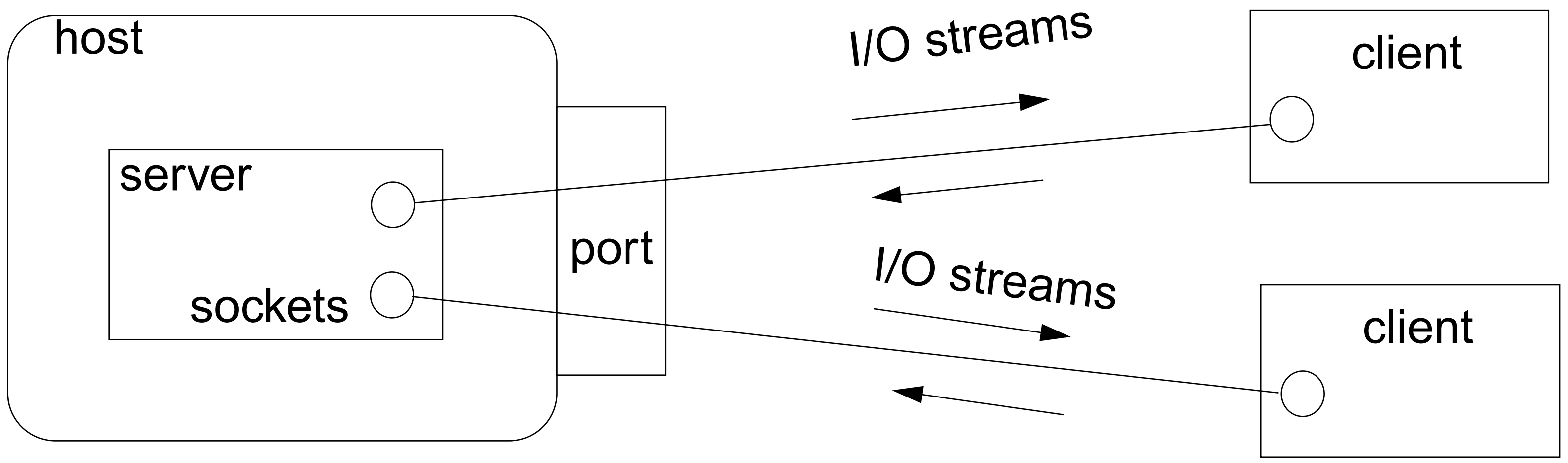


2 messages are sent on each **spanning tree edge**.

4 messages are sent on each **other edge**.

Connections via ports and sockets

Port:
- an **abstract connection point** of a computer; numerically encoded
- a **sever process** is determined to **respond to a certain port,** e. g. port 13: date and time
- client processes on other machines may send requests via **machine name and port number**



Socket:
- Abstraction of **network software** for communication via ports.
- Sockets are created from **machine address and port number**.
- **Several sockets** on one port may serve several clients.
- **I/O streams** can be setup on a socket.

Sockets and I/O-streams

Get a machine address:

```
InetAddress  addr1 = InetAddress.getByName ("java.sun.com"),
             addr2 = InetAddress.getByName ("206.26.48.100"),
             addr3 = InetAddress.getLocalHost();
```

Client side: create a socket that connects to the server machine:

```
Socket myServer = new Socket (addr2, port);
```

Setup I/O-streams on the socket:

```
BufferedReader in =
   new BufferedReader
      (new InputStreamReader (myServer.getInputStream()));

PrintWriter out =
   new PrintWriter (myServer.getOutputStream(), true);
```

Server side: create a specific socket, accept incoming connections:

```
ServerSocket listener = new ServerSocket (port);
...
Socket client = listener.accept(); ... client.close();
```

Worker paradigm

A task is decomposed dynamically in a **bag of subtasks**.
A set of **worker processes** of the same kind
solve subtasks of the bag and may **create new ones**.

Speedup if the processes are executed
in parallel on different processors.

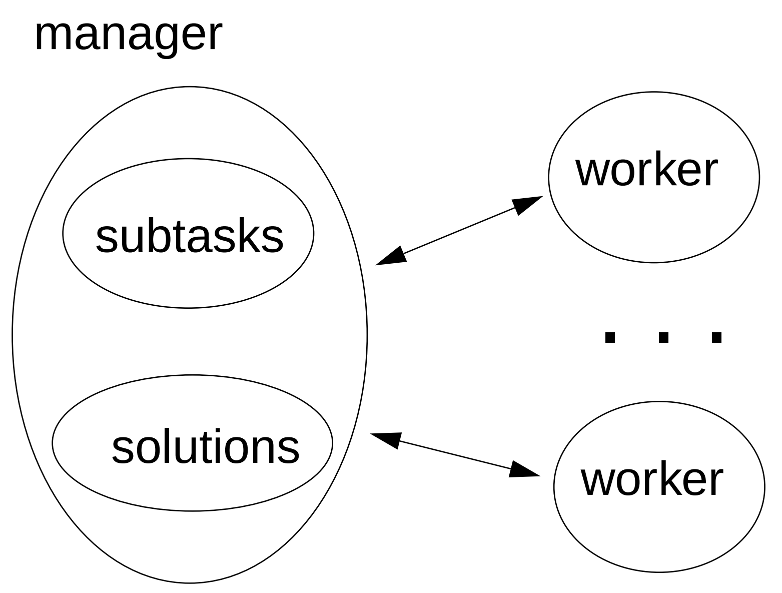
Applications: dynamically **decomposable** tasks, e.g.

- solving **combinatorial problems** with methods like Branch & Bound, Divide & Conquer, Backtracking
- image processing

general process structure:

manager process
manages the subtasks to be solved and
combines the solutions of the subtasks

worker process
solves one subtask after another,
creates new subtasks, and
provides solutions of subtasks.



Branch and Bound

Algorithmic method for the solution of **combinatorial problems** (e. g. traveling salesperson)

tree structured solution space is searched for a best solution

General scheme of operations:

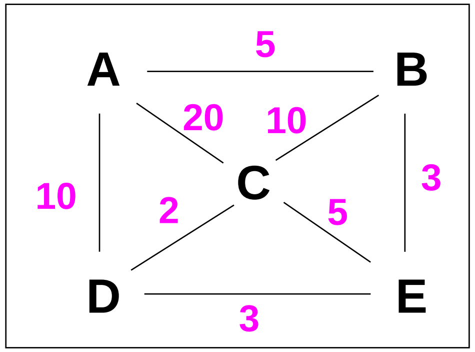
- **partial solution S is extended** to S_1, S_2, ... (e. g. add an edge to a path)
- is a partial solution **valid**? (e. g. is the added node reached the first time?)
- is S a **complete** solution? (e. g. are all nodes reached)
- **MinCost (S)** = C: each solution that can be created from S has at least cost C
(e. g. sum of the costs of the edges of S)
- **Bound**: costs of the best solution so far.

Data structures: a queue sorted according to MinCost; a bound variable

sequential algorithm:

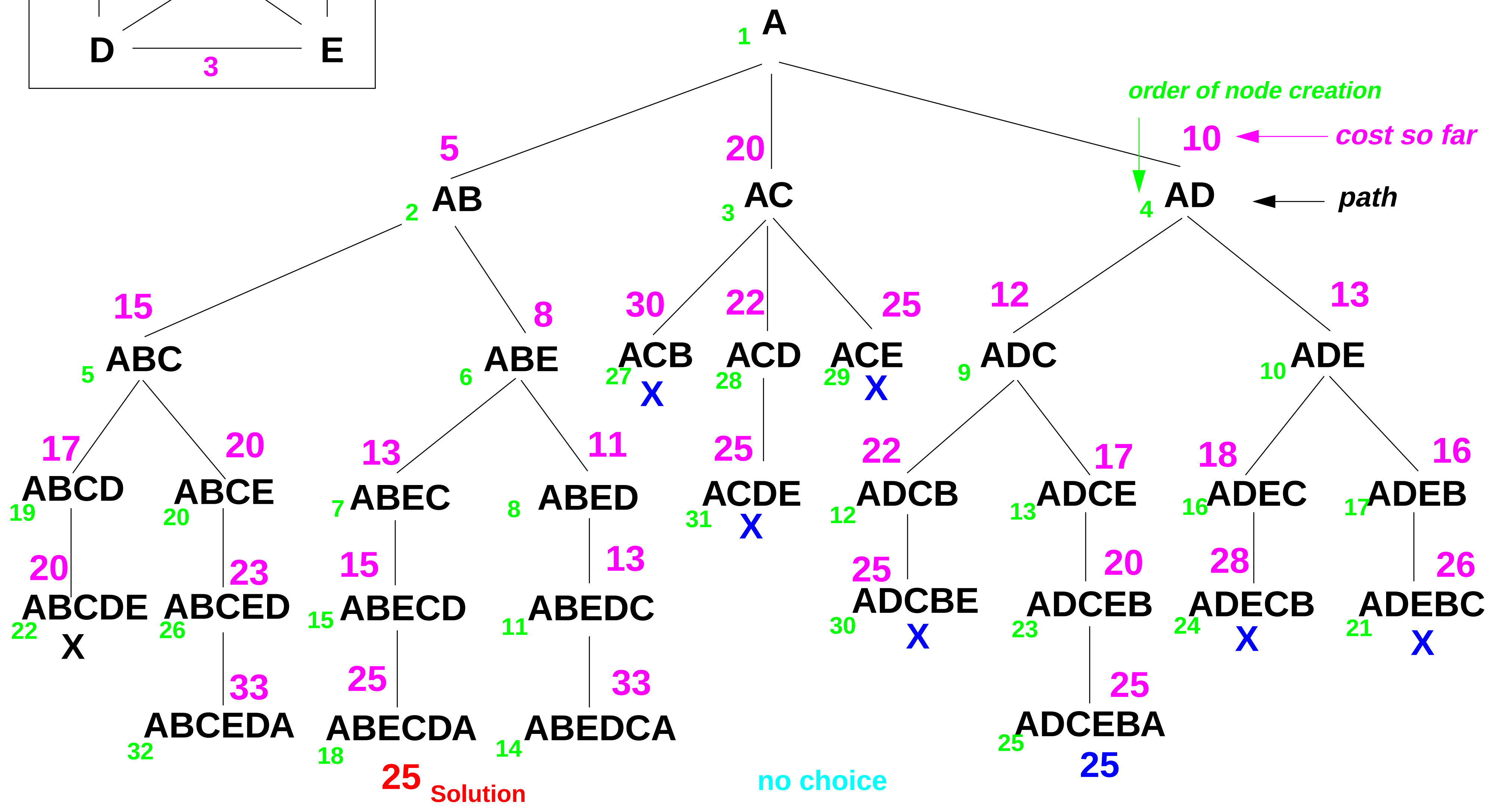
```
iterate until the queue is empty:
    remove the first element and extend it
    check the thus created new elements
    a new solution and a better bound may be found
    update the queue
```

B&B example: Travelling sales person



Connection graph

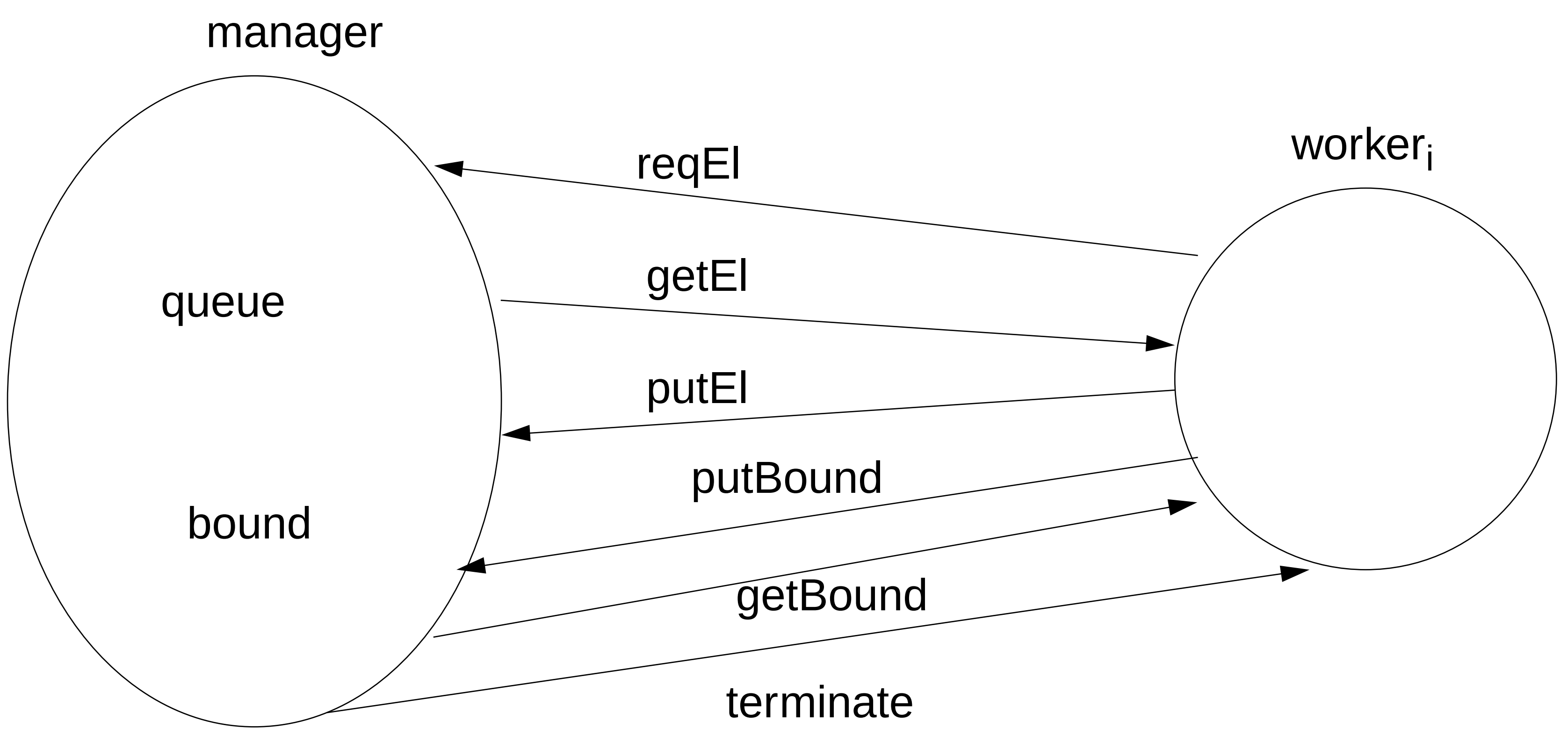
Solution space



Parallel Branch & Bound (central)

A **central manager process** holds the queue and the bound variable

Each **worker process** extends an element, checks it, computes its costs, and a new bound

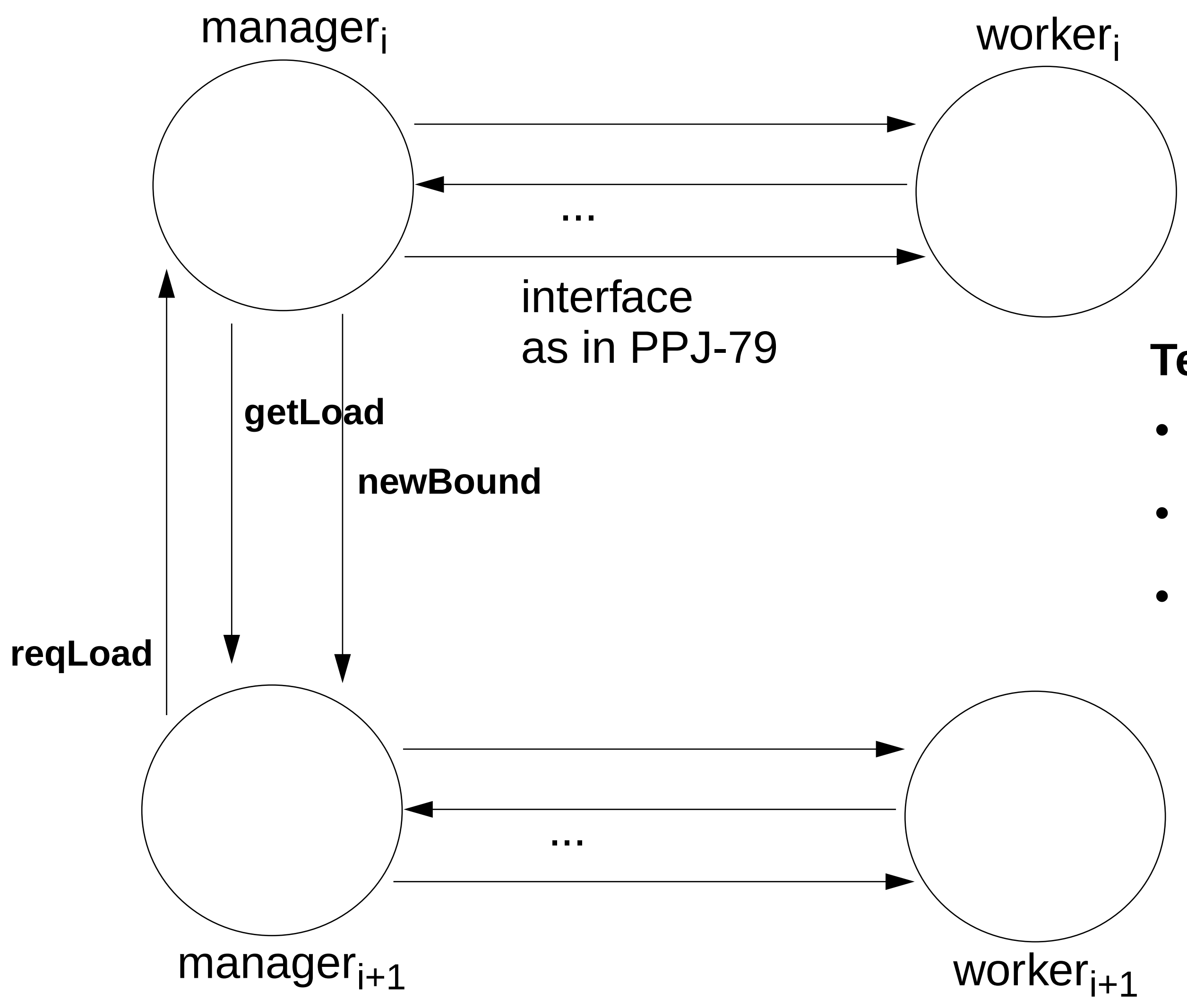


Protocol: reqEl (getEl [getBound] (putEl | putBound)* reqEl)* terminate
for a single Worker

Parallel Branch & Bound (distributed)

Several **manager processes cooperate** - one for each worker process.

The work load is balanced between neighbours, e. g. organized in a ring



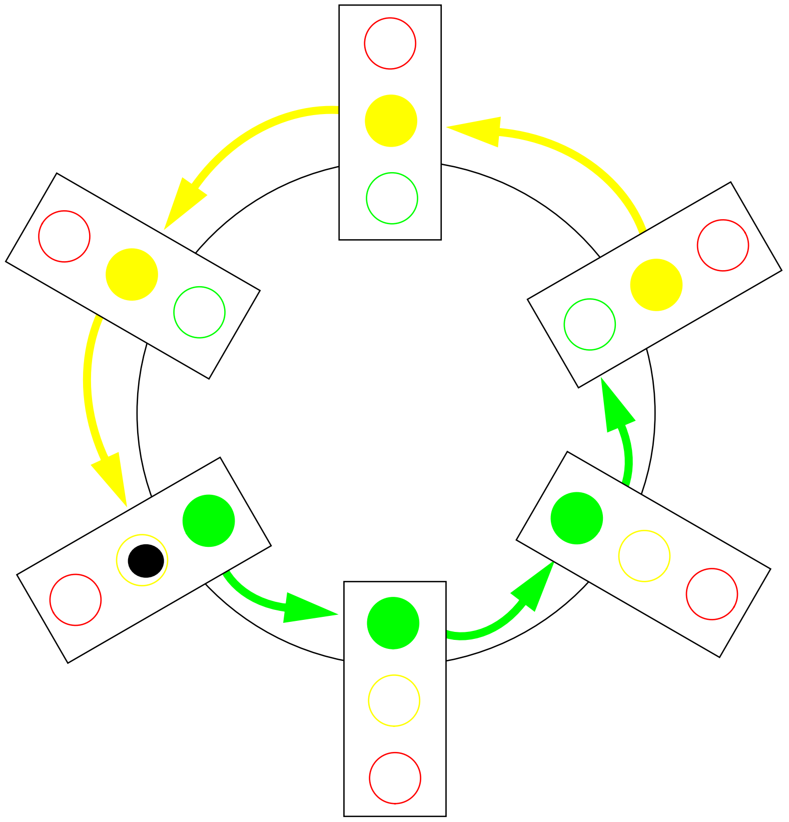
Termination condition:

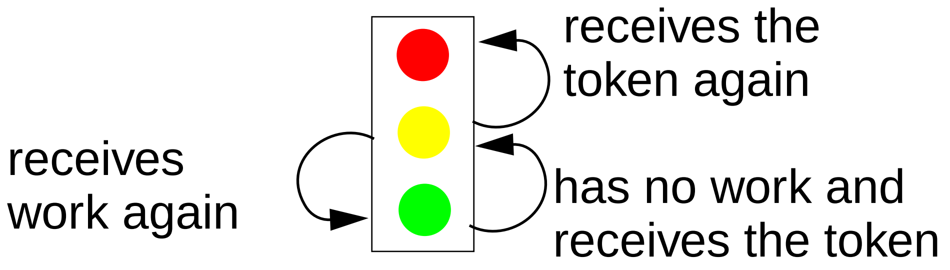
- all workers are inactive,
- no manager has another task
- all task channels are empty

Termination in a ring

Task: Determine a **global state of processes** that communicate in a **directed ring**, and inform all processes, e. g. „all processes are inactive“.

Idea: A token rotates through the ring and marks the processes (yellow) that have reached the state in question (inactive).
At the end of the marked sequence the mark may be reset again.
When the token reaches the end of the marked sequence, the state holds globally



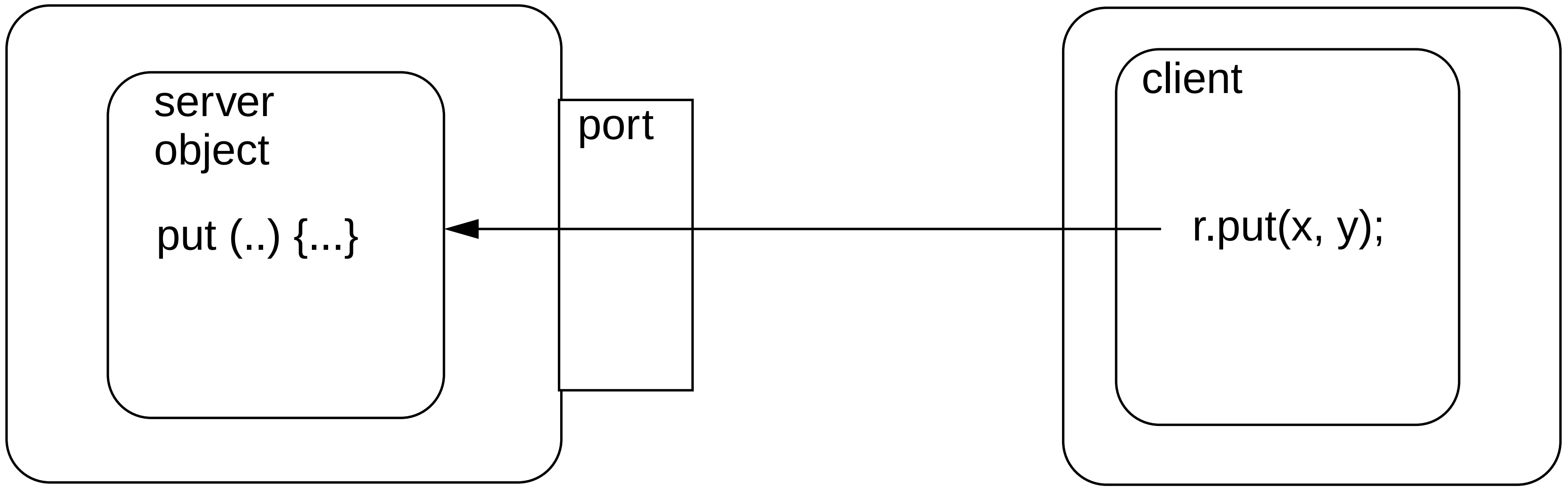


Method calls for objects on remote machines (RMI)

Remote Method Invocation (RMI): Call of a method for an object that is on a remote machine

In Java RMI is available via the library java.rmi.

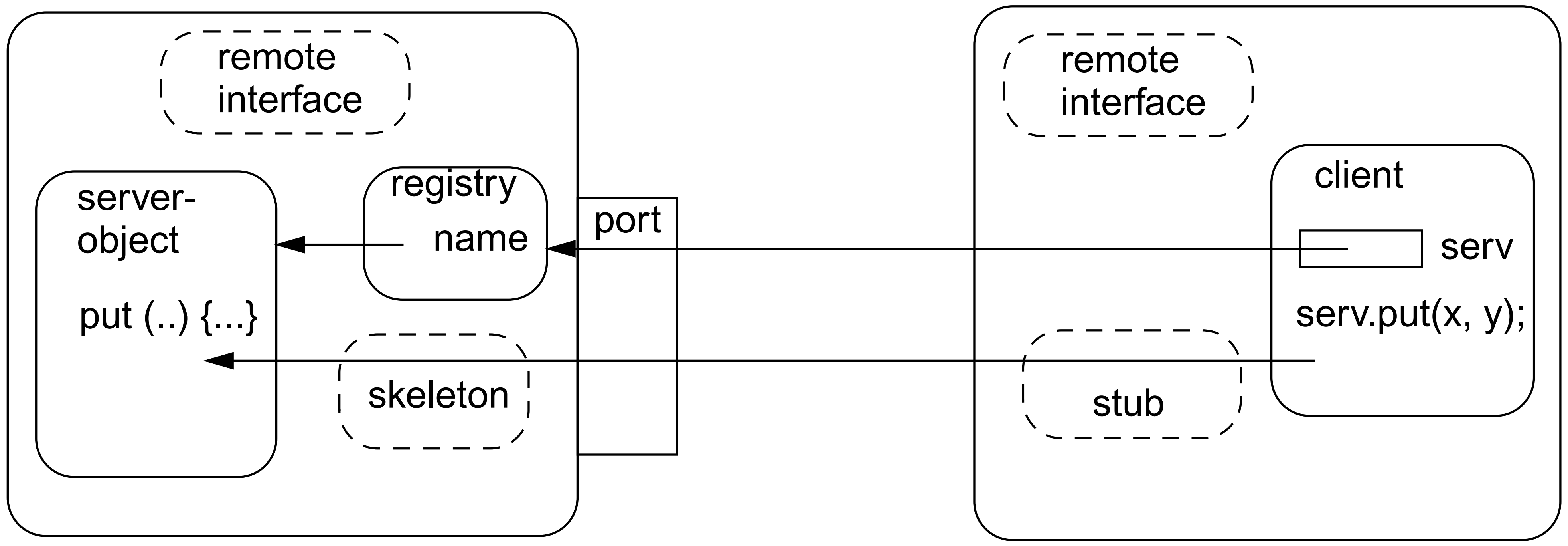
Comparable techniques: CORBA with IDL, Microsoft DCOM with COM



Tasks:

- **identify objects** across machine borders (object management, naming service)
- **interface** for remote accesses and executable proxies for the remote objects (skeleton, stub)
- **method call**, parameter and result are transferred (object serialization)

RMI in Java



remote interface: special requirements for interface methods

registry: system process for the machine and for a port;
establishes relations between names and object references

server skeleton: proxy of the server for remote accesses to server objects,
performs I/O transfer on the server side,

client stub: proxy of the server, performs I/O transfer on the client side

RMI development steps

Example: make a **`Hashtable`** available as a server object

1. Define a remote interface:
```
public interface RemoteMap extends java.rmi.Remote
{ public Object get (Object key) throws RemoteException; ...}
```
2. Develop an adapter class to adapt the server class to a remote interface:
```
public class RemoteMapAdapter extends UnicastRemoteObject
       implements RemoteMap
{ public RemoteMapAdapter (Hashtable a) { adaptee = a; }
  public Object get (Object key) throws RemoteException
  { return adaptee.get (key); }
  ...
}
```
3. Server main program creates the server object and enters it into the registry:
```
Hashtable adaptee = new Hashtable();
RemoteMapAdapter adapter = new RemoteMapAdapter (adaptee);
Naming.rebind (registeredObjectName, adapter);
```
4. Generate the skeleton and stub from the adapted server class;
copy the client stub on to the client machine:
```
rmic RemoteMapAdapter
```

RMI development steps (continued)

5. Client identifies the server object on a target machine and calls methods:
   ```
   Registry remoteRegistry = LocateRegistry.getRegistry (hostName);
   RemoteMap serv = (RemoteMap) remoteRegistry.lookup (remObjectName);
   v = serv.get (key);
   ```
6. Start a registry on the server machine:
   ```
   rmiregistry [port] &
   ```
   Default Port is 1099
7. Start some servers on the server machine.
8. Start some clients on client machines.

Objects as parameters of RMI calls

Parameters and results of RMI calls are transferred via I/O streams.

That is straight-forward for values of **basic types** and **strings**.

For objects in general:
The values of their variables are transferred,
on the receiver side a new object is created from those values.

The class of such objects has to implement the interface **`Serializable`**:

```
import java.io.Serializable;

class SIPair implements java.io.Serializable
{  private String s;
   private int i;

   public SIPair (String a, int b) { s = a; i = b; }
   public String toString () { return s + "-" + i; }
}
```