Communication between processes

Communication between processes

What problems emerge when communicating

- between separate address spaces
- between separate machines?

How do those environments differ from previous examples?

Recall that

- within a process, or with a shared virtual address space, threads can communicate naturally through ordinary data structures – object references created by one thread can be used by another
- failures are rare and usually occur at the granularity of whole processes
- OS-level protection is also performed at the granularity of processes

Communication between processes (2)

Most directly, introducing separate address spaces means that data is not directly shared between the threads involved

- At a low-level the representation of different kinds of data may vary between machines – e.g. big endian v little endian
- Names used may require translation – e.g. object locations in memory (at a low-level) or file names on a local disk (at a somewhat higher level)

More generally, we'll see four recurring problems in distributed systems:

- Components execute concurrently
- Components (and/or their communication channels) may fail independently
- Access to a 'global clock' cannot be assumed
- Inconsistent states can occur during operations (e.g. related changes to objects on different machines)

Communication between processes (3)

We'll look primarily at two different mechanisms for communication between processes

- Low-level communication using network sockets
  - ✔ A 'lowest-common-denominator': protocols like TCP are available on almost all platforms
  - ✘ Much more for the application programmer to think about; many wheels to re-invent
- Remote method invocation
  - ✔ Remote invocations look substantially like local calls: many low-level details are abstracted
  - ✘ Remote invocations look substantially like local calls: the programmer must remember the limits of this transparency and still consider problems such as independent failures
  - ✘ Not well suited to streaming or multi-casting data

Naming

How should processes identify which resources they wish to access?

Within a single address space in a Java program we could use object references to identify shared data structures and either

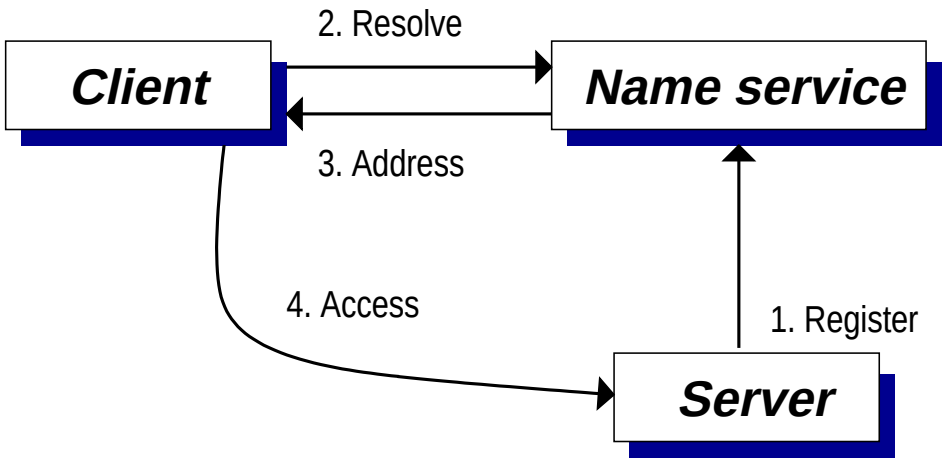
- pass them as parameters to a thread's constructor
- access them from static fields

When communicating between address spaces we need other mechanisms to establish

- unambiguously which item is going to be accessed
- where that item is located and how communication with it can be achieved

Late binding of names (e.g. `elite.cl.cam.ac.uk`) to addresses (`128.232.8.50`) is considered good practice – i.e. using a *name service* at run-time to resolve names, rather than embedding addresses directly in a program

Name services



How does the client now how to contact the name service?

- A *namespace* is a collection of names recognised by a name service – e.g. process IDs on one UNIX system, the filenames that are valid on a particular system or the Internet DNS names that are defined

- A *naming domain* is a section of a namespace operated under a single administrative authority – e.g. management of the `cl.cam.ac.uk` portion of the DNS namespace is delegated to the Computer Lab

- *Binding* or *name resolution* is the process of making a lookup on the name service

Name services (2)

Although we've shown the name service here as a single entity, in reality it may

- be *replicated* for availability (lookups can be made if any of the replicas are accessible) and read performance (lookups can be made to the nearest replica)
- be *distributed*, e.g. separate systems may manage different naming domains within the same namespace (updates to different naming domains require less co-ordination)
- allow *caching* of addresses by clients, or caching of partially resolved names in a hierarchical namespace

(See Part-II, Distributed Systems)

Names

Names are used to identify things and so they should be *unique* within the context that they are used. (A *directory service* may be used to select an appropriate name to look up – e.g. "find the nearest system providing service xyz")

When a namespace contains a single naming domain then simple unique IDs (UIDs) may be used – e.g. process IDs in UNIX

- UIDs are simply numbers in the range $0 \ldots 2^N - 1$ for an N-bit namespace. (Beware: UID $\neq$ user ID in this context!)

✔ Allocation is easy if N is large – just allocate successive integers

✘ Allocation is centralized (designs for allocating process IDs on highly parallel UNIX systems are still the subject of research)

✘ What can be done if N is small? When can/should UIDs be re-used?

Names (2)

More usually a *hierarchical* namespace is formed – e.g. filenames or DNS names

- ✔ The hierarchy allows *local allocation* if different allocators agree to use non-overlapping prefixes
- ✔ The hierarchy can often follow administrative *delegation* of control
- ✔ Locality of access within the structure may help implementation efficiency (if I lookup one name in `/usr/bin/` then perhaps I'm likely to lookup other names in that same directory)
- ✘ Lookups may be more complex. Can names be arbitrarily long?

Names (3)

We can also distinguish between *pure* and *impure* names

A pure name yields no information about the identified object – where it may be located or where its details may be held in a distributed name service. e.g. process IDs in UNIX

An impure name contains information about the object – e.g. e-mail to `tlh20@cam.ac.uk` will always be sent to a mail server in the University

- Are DNS names, e.g. `elite.cl.cam.ac.uk` pure or impure?
- Are IPv4 addresses, e.g. `128.232.8.50` pure or impure?

Names may have structure while still being pure – e.g. Ethernet MAC addresses are structured 48-bit UIDs and include manufacturer codes, and broadcast/multicast flags. This structure avoids centralized allocation

In other schemes, pure names may contain location *hints*. Crucially, impure names prevent the identified object from changing in some way (usually moving) without renaming

Protection

Require protection against unauthorised:

- release of information
  - reading or leaking data
  - violating privacy legislation
  - using proprietary software
  - covert channels
- modification of information
  - changing access rights
  - can do sabotage without reading information
- denial of service
  - causing a crash or intolerable load

How should access to resources be controlled?

- When a system is built from multiple processes
- ...when these may be executing on different systems
- ...when some may be operating as servers on behalf of many clients

Protection (2)

- Some other protection mechanisms:
  - lock the computer room (prevent people from tampering with the hardware)
  - restrict access to system software
  - de-skill systems operating staff
  - keep designers away from final system!
  - use passwords (in general challenge/response)
  - use encryption
  - legislate

- ref: Saltzer + Schroeder Proc. IEEE, Sept 75
  - design should be public
  - default should be no access
  - check for current authority
  - give each process minimum possible authority
  - mechanisms should be simple, uniform and built in to lowest layers
  - should be psychologically acceptable
  - cost of circumvention should be high
  - minimize shared access

Access matrix

Access matrix is a matrix of subjects against objects.

Subject (or principal) might be:

- users e.g. by system user ID
- executing process in a protection domain
- sets of users or processes

Objects are things like:

- files
- devices
- domains / processes
- message ports (in microkernels)

Matrix is large and sparse ⇒ don't want to store it all.

Two common representations:

1. by object: store list of subjects and rights with each object ⇒ *access control list*

2. by subject: store list of objects and rights with each subject ⇒ *capabilities*

Access control lists

Often used in storage systems:

- system naming scheme provides for ACLs to be inserted at each level of a hierarchical name, e.g. files
- if ACLs stored on disk, check is made in software ⇒ must only use on low duty cycle
- for higher duty cycle must cache results of check
- e.g. Multics: open file = memory segment.
  On first reference to segment:
  1. interrupt (segment fault)
  2. check ACL
  3. set up segment descriptor in segment table
- most systems check ACL
  - when file opened for read or write
  - when code file is to be executed
- access control by program, e.g. Unix
  - exam prog, RWX by examiner, X by student
  - data file, A by exam program, RW by examiner

Capabilities

Capabilities associated with active subjects, so:

- store in address space of subject
- must make sure subject can't forge capabilities
- easily accessible to hardware
- can be used with high duty cycle
  e.g. as part of addressing hardware
  - Plessey PP250
  - CAP I, II, III
  - IBM system/38
  - Intel iAPX432
- have special machine instructions to modify (restrict) capabilities
- support passing of capabilities on procedure call

Can also use *software* capabilities. Checked by encryption. Nice for distributed systems

Capabilities (2)

Tagged Architectures (e.g. IBM system/38):

- all words in memory and the processor registers are tagged as containing either data or a capability
- tag stays with contents on all copy operations
- system checks ALU operations for validity

Capability segments (e.g. CAP):

- capabilities for code segment held in special capability segment
- only a restricted set of operations are allowed on capability segments
- provide a cache of entries in capability segments in special capability registers
- use associative store, per domain capability list, central capability list
- add *enter* capability

Software schemes (e.g. EROS)

- require capabilities for all system services
- fake out `enter` via IPC.

Capabilities (3)

- Capabilities nice for distributed systems but:
  - messy for application, and
  - revocation is tricky.
- Could use timeouts (e.g. Amoeba).
- Alternatively: combine passwords and capabilities.
- Store ACL with object, but key it on capability (not implicit concept of “principal” from OS).
- Advantages:
  - revocation possible
  - multiple “roles” available.
- Disadvantages:
  - still messy (use ‘implicit’ cache?).

Facilities in Java

We'll now look at how these techniques apply to Java applications

Within Java applications object references can be used as unforgeable capabilities, e.g. when running multiple applets within a single JVM

- Access modifiers on constructors prevent arbitrary instantiation of classes
- Access control checks can be performed at instantiation time and – if these fail – instantiation can be aborted by throwing an exception

For many kinds of access the *security manager* provides a mechanism for enforcing simple controls

- A security manager is implemented by `java.lang.SecurityManager` (or a sub-class)
- An instance of this is installed using `System.setSecurityManager(...)` (itself an operation under the control of the current security manager)

Facilities in Java (2)

Most checks are made by delegating to a `checkPermission` method, e.g. for dynamically loading a native library

```
checkPermission(
   new RuntimePermission(
      "loadLibrary."+lib));
```

Decisions made by `checkPermission` are relative to a particular *security context*. The current context can be obtained by invoking `getSecurityContext` and checks then made on behalf of another context

Permissions can be granted in a policy definition file, passed to the JVM on the command line with `-Djava.security.policy=`*filename*

```
grant {
  permission java.net.SocketPermission
          "*:1024-65535", "connect,accept";
};
```

`http://java.sun.com/products/jdk/1.2/docs/guide/security/index.html`

Low-level communication

Two basic network protocols are available in Java: datagram-based UDP and stream-based TCP (see Digital Communication I)

Communication occurs between UDP *sockets* which are addressed by giving an appropriate IP address and a UDP port number (0..65535, although 0 not accessible through common APIs, 1..1023 reserved for privileged use)

UDP sockets provide *unreliable datagram-based* communication that is subject to:

- *Loss*: datagrams that are sent may never be received, and
- *Re-ordering*: datagrams are forwarded separately within the network and may arrive out of order

A checksum is used to guard against corruption (corrupt data is discarded by the protocol implementation and the application perceives it as loss)

The *framing* within datagrams is preserved – e.g. if fragmentation occurs within the network

Low-level communication (2)

Naming is handled by

- Using the DNS to map textual names into IP addresses, `InetAddress.getByName("elite.cl.cam.ac.uk")`
- Using 'well-known' port numbers for particular UDP services which wish to be accessible to clients (See the `/etc/services` file on a UNIX system)

UDP sockets are represented by instances of `java.net.DatagramSocket`. The 0-argument constructor creates a new socket that is bound to an available port on the local host machine. This identifies the *local* endpoint for the communication

Datagrams are represented in Java as instances of `java.net.DatagramPacket`. The most elaborate constructor

```
DatagramPacket(byte buf[], int length,
  InetAddress address, int port)
```

specifies the data to send (`length` bytes from within `buf`) and the destination `address` and `port`

UDP example

```
import java.net.*;

public class Send {
  public static void main (String args[]) {
    try {
      DatagramSocket s = new DatagramSocket ();
      byte[]         b = new byte[1024];
      int            i;

      for (i = 0; i < args.length - 2; i ++)
        b[i] = Byte.parseByte (args[2 + i]);

      DatagramPacket p = new DatagramPacket (
            b, i,
            InetAddress.getByName (args[0]),
            Integer.parseInt (args[1]));

      s.send(p);

    } catch (Exception e) {
      System.out.println("Caught " + e);
    }
  }
}
```

UDP example (2)

```
import java.net.*;

public class Recv {
  public static void main (String args[]) {
    try {
      DatagramSocket s = new DatagramSocket ();
      byte[]         b = new byte[1024];
      DatagramPacket p =
                  new DatagramPacket (b, 1024);

      System.out.println("Port: " +
                  s.getLocalPort());

      s.receive(p);

      for (int i = 0; i < p.getLength (); i ++)
        System.out.print ("" + b[i] + " ");

      System.out.println ("\nFrom: " +
        p.getAddress () + ":" + p.getPort ());
    } catch (Exception e) {
      System.out.println("Caught " + e);
    }
  }
}
```

Problems using UDP

Many facilities must be implemented manually by the application programmer:

- ✘ Detection and recovery from loss
- ✘ Flow control (preventing the receiver from being swamped with too much data)
- ✘ Congestion control (preventing the network from being overwhelmed)
- ✘ Conversion between application data structures and arrays of bytes (*marshaling*)

Of course, there are situations where UDP is directly useful

- ✔ Communication with existing UDP services (e.g. some DNS name servers)
- ✔ Broadcast and multicast are possible (e.g. address 255.255.255.255 ⇒ all machines on the local network

TCP sockets

The second basic form of inter-process communication is provided by TCP sockets

Naming is again handled using the DNS and well-known port numbers as before. There is no relationship between UDP and TCP ports having the same number

TCP provides a reliable bi-directional connection-based byte-stream with flow control and congestion control

What doesn't it do?

- Unlike UDP the interface exposed to the programmer is not datagram based: framing must be provided explicitly
- Marshaling must still be done explicitly – but serialization may help here
- Communication is one-to-one

In practice TCP forms the basis for many internet protocols – e.g. FTP and HTTP are both currently deployed over it

TCP sockets (2)

Two principal classes are involved in exposing TCP sockets in Java:

- `java.net.Socket` represents a connection over which data can be sent and received. Instantiating it directly initiates a connection from the current process to a specified address and port. The constructor blocks until the connection is established (or fails with an exception)

- `java.net.ServerSocket` represents a socket awaiting incoming connections. Instantiating it starts the local machine listening for connections on a particular port. `ServerSocket` provides an `accept` operation that blocks the caller until an incoming connection is received. It then returns an instance of `Socket` representing that connection

The system will usually buffer only a small (5) number of incoming connections if `accept` is not called

Typically programs that expect multiple clients will have one thread making calls to `accept` and starting further threads for each connection

TCP example

```
import java.net.*;
import java.io.*;

public class TCPSend {
  public static void main (String args[]) {
    try {
      Socket s = new Socket (
          InetAddress.getByName (args[0]),
          Integer.parseInt (args[1]));

      OutputStream os = s.getOutputStream ();

      while (true) {
        int i = System.in.read();
        os.write(i);
      }

    } catch (Exception e) {
      System.out.println("Caught " + e);
    }
  }
}
```

TCP example (2)

```
import java.net.*;
import java.io.*;

public class TCPRecv {
  public static void main (String args[]) {
    try {
      ServerSocket serv = new ServerSocket (0);
      System.out.println ("Port: " +
              serv.getLocalPort ());
      Socket           s = serv.accept ();
      System.out.println ("Remote addr: " +
              s.getInetAddress());
      System.out.println ("Remote port: " +
              s.getPort());
      InputStream    is = s.getInputStream ();
      while (true) {
        int i = is.read ();
        if (i == -1) break;
        System.out.write (i);
      }
    } catch (Exception e) {
      System.out.println("Caught " + e);
    }
  }
}
```

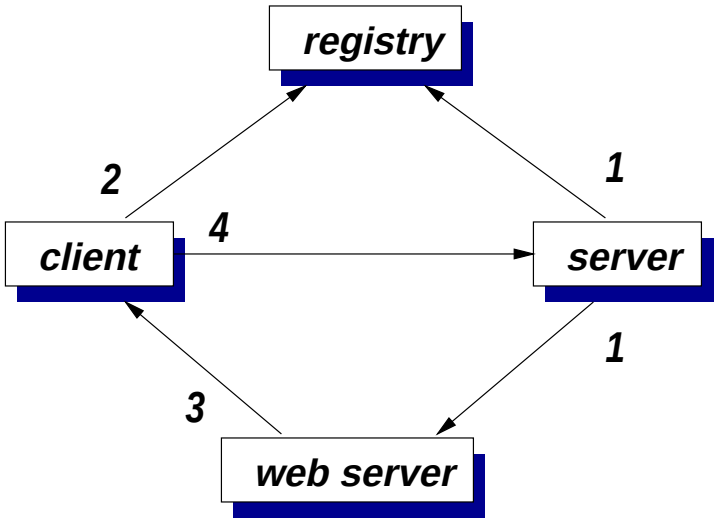
Remote method invocation

Using UDP or TCP it was necessary to

- Decide how to represent data being sent over the network – either packing it into arrays of bytes (in a `DatagramPacket`) or writing it into an OutputStream (using a `Socket`)
- Use a rather inflexible naming system to identify servers – updates to the DNS may be difficult, access to a specific port number may not always be possible
- Distribute the code to all of the systems involved and ensure that it remains consistent
- Deal with failures (e.g. the remote machine crashing – something a 'reliable' protocol like TCP cannot mask)

Java RMI presents a higher level interface that addresses some of these concerns. Although it is remote *method* invocation, the principles are the same as for remote *procedure* call (RPC) systems

Remote method invocation (2)



1. A server registers a reference to a remote object with the *registry* (a basic name service) and deposits associated `.class` files with a web server

2. A client queries the registry to obtain a reference to a remote object

3. The client obtains the `.class` files needed to access the remote object from a web server (if they are not already available locally)

4. The client makes an RMI call to the remote object

The registry acts here as a name service, holding names of the form `//thor.cam.ac.uk/tlh20-example-1.2`

Remote method invocation (3)

Parameters and results are generally passed by making *deep copies* when passed or returned over RMI

- i.e. copying proceeds recursively on the object passed, objects reachable from that etc (→ take care to reduce parameter sizes)
- The structure of object graphs is preserved – e.g. data structures may be cyclic
- Remote objects are passed *by reference* and so both caller and callee will interact with *the same* remote object if a reference to it is passed or returned

Note that Java only supports remote *method* invocation – changes to fields must be made using `get`/`set` methods

Other implementation choices:

- Perform a shallow copy and treat other objects reachable from that as remote data (as above, would be hard to implement in Java) or copy them incrementally
- Emulate 'pass by reference' by passing back any changes with the method results (what about concurrent updates?)

RMI - Interfaces

Suppose that we wish to define a simple remote object on which a single method `spell` is defined:

```
1  package tlh20.rmi;
2
3  import java.rmi.*;
4
5  public interface Phonetic extends Remote {
6
7    public final static String URL =
8        "//thor.cam.ac.uk/tlh20-example-1.2";
9
10   public String [] spell (String s)
11       throws RemoteException;
12 }
```

- All RMI invocations are made across *remote interfaces* extending `java.rmi.Remote`
- The field URL in Lines 7–8 will be used to name a particular remote object implementing this interface. It's included here for easy access by both client and server
- All remote methods must throw `RemoteException`

RMI - Client

```
1  package tlh20.rmi;
2
3  import java.rmi.*;
4
5  public class PhoneticClient {
6
7    public static void main (String [] args) {
8      try {
9        System.setSecurityManager (
10             new RMISecurityManager ());
11
12       Phonetic p = (Phonetic)
13            Naming.lookup (Phonetic.URL);
14
15       String [] results = p.spell ("Example");
16
17       for (int r = 0; r < results.length; r++)
18         System.out.println (results [r]);
19     }
20     catch (Exception e) {
21       System.out.println ("Exception: " + e);
22     }
23   }
24 }
```

RMI - Client (2)

Note how few differences there are in the client compared with local invocations on an instance of a class implementing `Phonetic`:

- The security manager installed in lines 9–10 is an example one for use by RMI applications that use downloaded code
- Lines 12–13 obtain an instance of a class implementing the `Phonetic` interface. Invocations on this instance will be made on a remote object registered under the name `Phonetic.URL`
- The exception handler in lines 20–22 may see
  - `NotBoundException` – no remote object has been associated with the name `Phonetic.URL`
  - `RemoteException` – if the RMI registry could not be contacted (12–13) or if there was a problem with the call (15)
  - `AccessException` – if the operation has not been permitted

RMI - Server

```
1  package tlh20.rmi;
2
3  import java.net.*;
4  import java.rmi.*;
5  import java.rmi.server.*;
6
7  public class PhoneticServer
8       extends UnicastRemoteObject
9       implements Phonetic
10 {
11   public static void main (String [] args) {
12     try {
13       System.setSecurityManager (
14             new RMISecurityManager ());
15
16       PhoneticServer s = new PhoneticServer ();
17
18       Naming.rebind (Phonetic.URL, s);
19       System.out.println (Phonetic.URL +
20             " server running");
21     }
22     catch (Exception e) {
23       System.out.println ("Exception: " + e);
24     };
25   }
```

RMI - Server (2)

```
26  public PhoneticServer () throws RemoteException {
27    super ();
28  }
29
30  private final static String [] WORDS = { "alfa",
31  "bravo", "charlie", "delta", "echo", "foxtrot",
32  "golf", "hotel", "India", "Juliet", "kilo",
33  "Lima", "Mike", "November", "Oscar", "papa",
34  "Quebec", "Romeo", "sierra", "tango", "uniform",
35  "victor", "whiskey", "x-ray", "yankee", "zulu" };
36
37  public String [] spell (String s)
38        throws RemoteException
39  {
40    String source = s.toUpperCase ();
41    String [] reply = new String [s.length ()];
42    for (int i = 0; i < s.length (); i++) {
43      try {
44        int w = (int) source.charAt (i) - (int) 'A';
45        reply [i] = WORDS [w];
46      }
47      catch (Exception e) {reply [i] = "?";}
48    }
49    return reply;
50  }
```

Putting it all together

- Compile the remote interface class, client and server:

```
$ javac tlh20/rmi/Phonetic.java
$ javac tlh20/rmi/PhoneticClient.java
$ javac tlh20/rmi/PhoneticServer.java
```

- Generate stub classes from the server:

```
$ export PUBCLASSES=/home/tlh20/\
public_html/java/classes/

$ rmic -v1.2 -d $PUBCLASSES \
tlh20.rmi.PhoneticServer
```

- Generate a security policy file:

```
grant {
  permission java.net.SocketPermission
      "*:1024-65535", "connect,accept";
  permission java.net.SocketPermission
      "*:80", "connect";
  permission java.util.PropertyPermission
      "java.rmi.server.codebase", "read";
  permission java.util.PropertyPermission
      "user.name", "read,write";
};
```

Putting it all together (2)

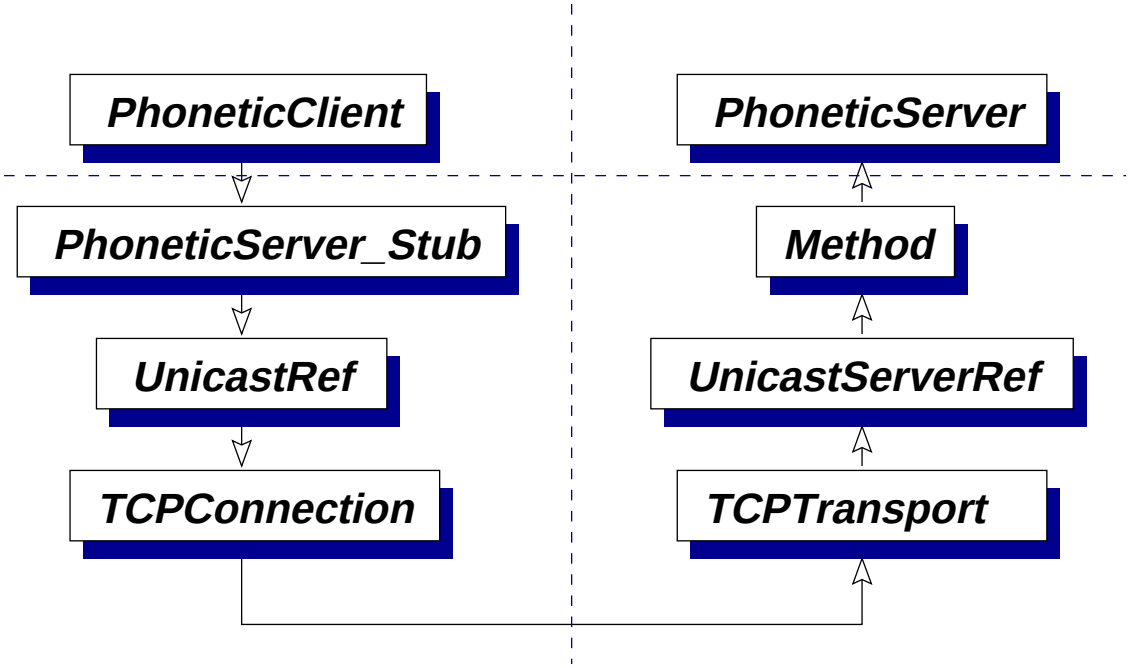
- Start the server running:

```
$ export CODEBASE=http://hammer.thor.cam.ac.uk\
/~tlh20/java/classes/

$ java -Djava.rmi.server.codebase=$CODEBASE \
     -Djava.security.policy=security.policy \
     tlh20.rmi.PhoneticServer

//thor.cam.ac.uk/tlh20-example-1.2 server running
```

- Start the client running:

```
$ java -Djava.security.policy=security.policy \
       tlh20.rmi.PhoneticClient
echo
x-ray
alfa
Mike
papa
Lima
echo
```

RMI implementation



- The `_Stub` class is the one created by the `rmic` tool – it transforms invocations on the `Phonetic` interface into generic invocations of an `invoke` method on `UnicastRef`
- `UnicastRef` is responsible for selecting a suitable network transport for accessing the remote object – in this case TCP
- `UnicastServerRef` uses the ordinary reflection interface to dispatch calls to remote objects

RMI implementation (2)

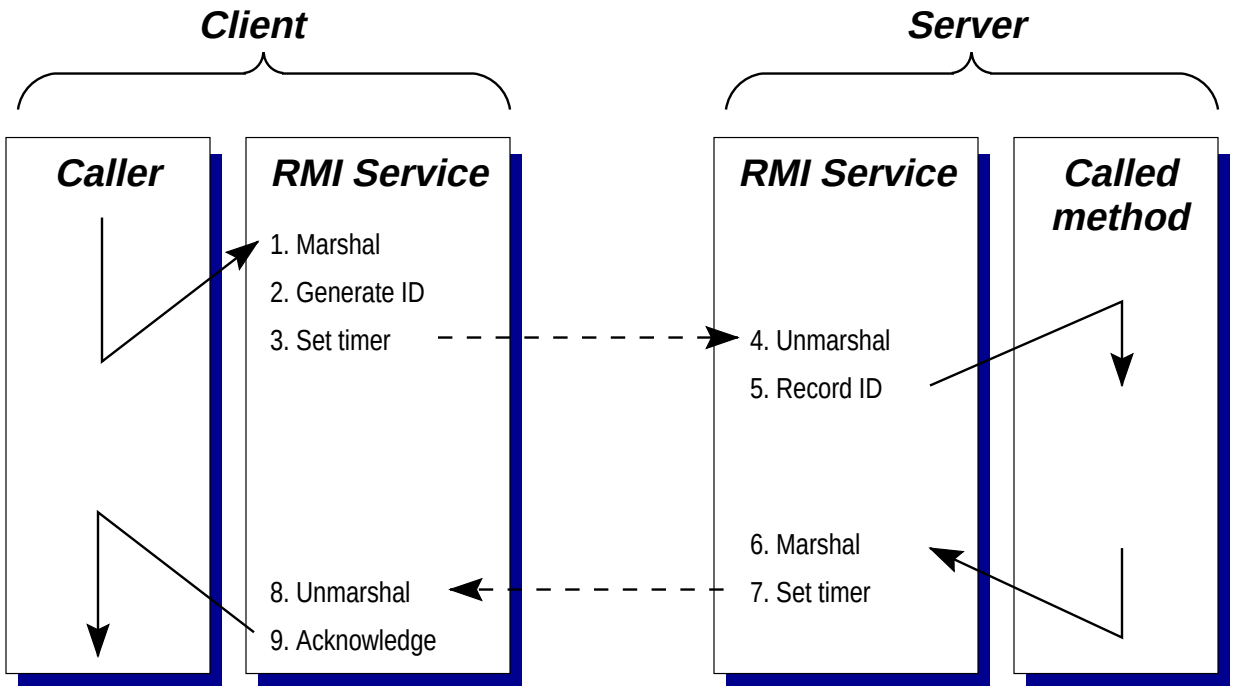
With the TCP transport RMI creates a new thread on the server for each incoming connection that is received

- A remote objects should be prepared to accept concurrent invocations of its methods
- Remember: the `synchronized` modifier applies to a method's *implementation*. It must be applied to the definition in the server class, not the interface

✔ This avoids deadlock if remote object A invokes an operation on remote object B which in turn invokes an operation on A

✘ The application programmer must be aware of how many threads might be created and the impact that they may have on the system

RMI implementation (3)



What could be done without TCP?

We need to manually implement:

- Reliable delivery of messages subject to loss in the network
- Association between invocations and responses – shown here using per-call RPC identifier with which all messages are tagged

RMI implementation (4)

Even this simple protocol requires multiple threads: e.g. to re-send lost acknowledgements after the client-side RMI service has returned to the caller

What happens if a timeout occurs at 3? Either the message sent to the server was lost, or the server failed before replying

- *At-most-once* semantics ⇒ return failure indication to the application
- *'Exactly'-once* semantics ⇒ retry a few times with the same RPC id (so server can detect retries)

What happens if a timeout occurs at 7? Either the message sent to the client was lost, or the client failed

No matter what is done, the client cannot distinguish, on the basis of these messages, server failures before / after making some change to persistent storage

Defining remote interfaces

Recall that with Java RMI the interface to a remote object is defined as an ordinary `interface` that extends `java.rmi.Remote`

✔ Easy to use in Java-based systems

✘ What about interoperability with other languages?

Java RMI is rather unusual in using ordinary language facilities to define remote interfaces. Usually a specific *Interface Definition Language* (IDL) is used

- This acts as a 'lowest common denominator' presenting features common to many languages
- The IDL has *language bindings* that define how its features are realized in a particular language
- An IDL compiler generates per-language stubs (contrast with the `rmic` tool that only generates stubs for the JVM)

OMG IDL

We'll take OMG IDL (used in CORBA) as a typical example

```
1  //POS Object IDL example
2  module POS {
3    typedef string Barcode;
4
5    interface InputMedia {
6      typedef string OperatorCmd;
7      void barcode_input(in Barcode item);
8      void keypad_input(in OperatorCmd cmd);
9    };
10 };
```

- A `module` defines a namespace within which a group of related type definitions and interface definitions occur
- Interfaces can be derived using multiple inheritance
- Built-in types include basic integers (e.g. `long` holding $-2^{31} \ldots 2^{31} - 1$ and `unsigned long` holding $0 \ldots 2^{32} - 1$), floating point types, 8-bit characters, `boolean`s and `octet`s
- Parameter modifiers `in`, `out` and `inout` define the direction in which parameters are copied

OMG IDL (2)

Type constructors allow structures, discriminated unions, enumerations and sequences to be defined:

```
struct Person {
  string name;
  short age;
};

union Result switch(long) {
  case 1 : ResultDataType r;
  default : ErrorDataType e;
};

enum Color { red, green, blue };

typedef sequence<Person> People;
```

Interfaces can define *attributes* (unlike Java interfaces), but these are just shorthand for pairs of method definitions:

```
attribute long value;
```

$\rightarrow$

```
long _get_value();
void _set_value(in long v);
```

OMG IDL (3)

IDL construct	*Java construct*
`module`	`package`
`interface`	`interface` + classes
`constant`	`public static final`
`boolean`	`boolean`
`char`, `wchar`	`char`
`octet`	`byte`
`string`, `wstring`	`java.lang.String`
`short`	`short`
`unsigned short`	`short`
`long`	`long`
`unsigned long`	`long`
`float`	`float`
`double`	`double`
`eunm`, `struct`, `union`	`class`
`sequence`, `array`	`array`
`exception`	`class`
`readonly attribute`	Read-accessor method
`attribute`	Read,write-accessor methods
`operation`	Method

- 'Holder classes' are used for `out` and `inout` parameters – these contain a field appropriate to the type of the parameter

Microsoft .NET

Instead of defining a separate IDL and per-language bindings, the Microsoft .NET platform defines a *common language subset* and programming conventions for making definitions that conform to it

Many familiar features: static typing, objects (classes, fields, methods, properties), overloading, single inheritance of implementations, multiple implementation of interfaces, . . .

Metadata describing thse definitions is available at run-time, e.g. to control marshaling

- Interfaces can be defined in an ordinary programming language and do not need an explicit IDL compiler
- Languages vary according to whether they can be used to write clients or servers in this system – e.g. JScript and COBOL vs VB, C#, SML


Transactions

Transactions

We've now seen mechanisms for

- Controlling concurrent access to objects
- Providing access to remote objects

Using these facilities correctly, and particularly in combination, is extremely difficult. What improved abstractions could be provided?

Ideally the programmer may wish to write something like

```
transactionally {
  if (source.balance() >= amount) {
    source.withdraw (amount);
    destination.deposit (amount);
    return true;
  } else {
    return false;
  }
}
```

Transactions (2)

The intent is that code within a `transactionally` block will execute without interference from other activities, in particular

- other operations on the same objects
- system crashes

We'll say that a transaction either commits (i.e. succeeds) or aborts (i.e. fails).

Of course, we can't provide complete resilience to system crashes, but we can say that

- if enough of the system keeps working
- then the results of committed transactions are not lost
- and the effects of non-committed transactions are not seen

Transactions (3)

In more detail we'd like committed transactions to satisfy four 'ACID' properties:

Atomicity – either all or none of the transaction's operations are performed

— programmers do not have to worry about 'cleaning up' after a transaction aborts; the system ensures that it has no visible effects

Consistency – a transaction transforms the system from one consistent state to another

— essentially the transaction must be implemented to preserve desired invariants, e.g. totals across accounts

Isolation – the effects of a transaction are not visible to other transactions until it is committed

— in the strictest case, another transaction shouldn't read the `source` and `destination` amounts mid-transfer

Durability – the effects of committed transactions endure subsequent system failures

— when the system confirms the transaction has committed it must ensure any changes will survive faults

Transactions (4)

These requirements can be grouped into two categories:

- Atomicity and durability refer to the persistence of transactions across system failures.

  We want to ensure that no 'partial' transactions are performed (atomicity) and we want to ensure that system state does not regress by apparently-committed transactions being lost (durability)

- Consistency and isolation concern ensuring correct behaviour in the presence of concurrent transactions

  As we'll see there are trade-offs between the ease of programming within a particular transactional framework, the extent that concurrent execution of transactions is possible and the isolation that is enforced

Persistent storage

Assume a *fail-stop* model of crashes in which

- the contents of main memory (and above in the memory hierarchy) is lost
- non-volatile storage is preserved (e.g. data written to disk)

$\Rightarrow$ if we want the state of an object to be preserved across system failures then we must either

- ensure that sufficient replicas exist on different machines that the risk of losing all is tolerable (*Part-II Distributed Systems*)
- ensure that the enough information is written to non-volatile storage in order to recover the state after a restart

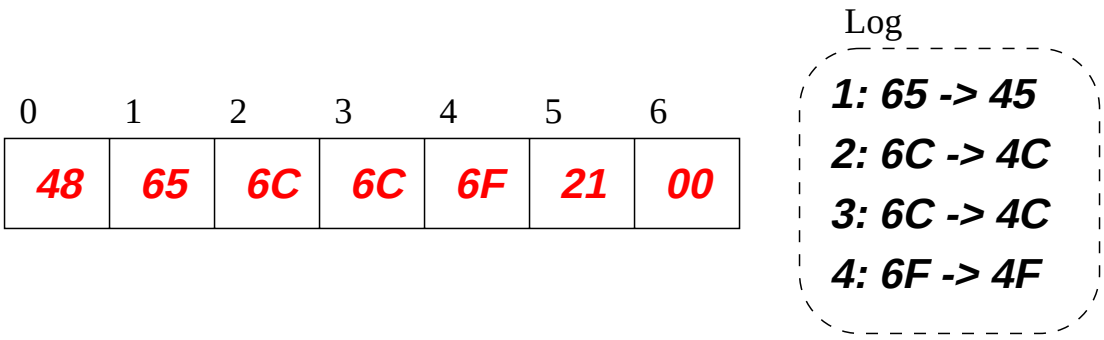
Can we just write object state to disk before every commit? (e.g. invoking `flush()` on any kind of Java `OutputStream`)

✘ Not directly: the failure may occur part-way through the disk write (particularly for large amounts of data)

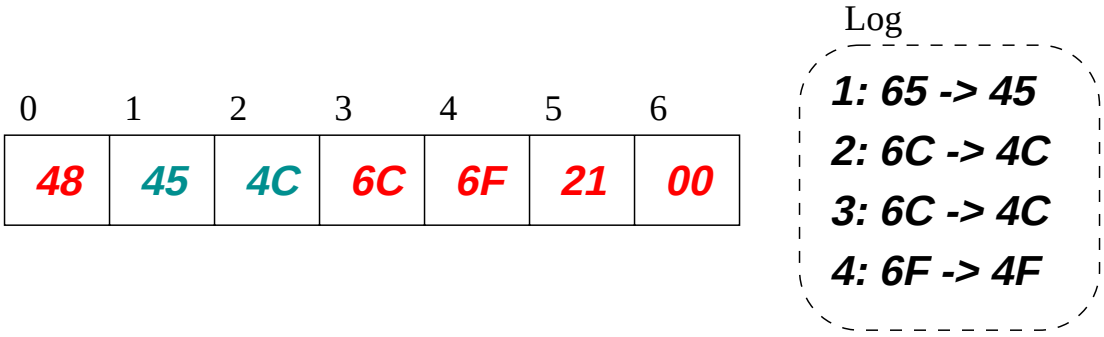
Persistent storage – logging

We could split the update into stages:

1. Write details of the proposed update to an *write-ahead log* – e.g. in a simple case giving the old and new values of the data, or giving a list of smaller updates as a set of $(address, old, new)$ tuples



2. Proceed through the log making the updates



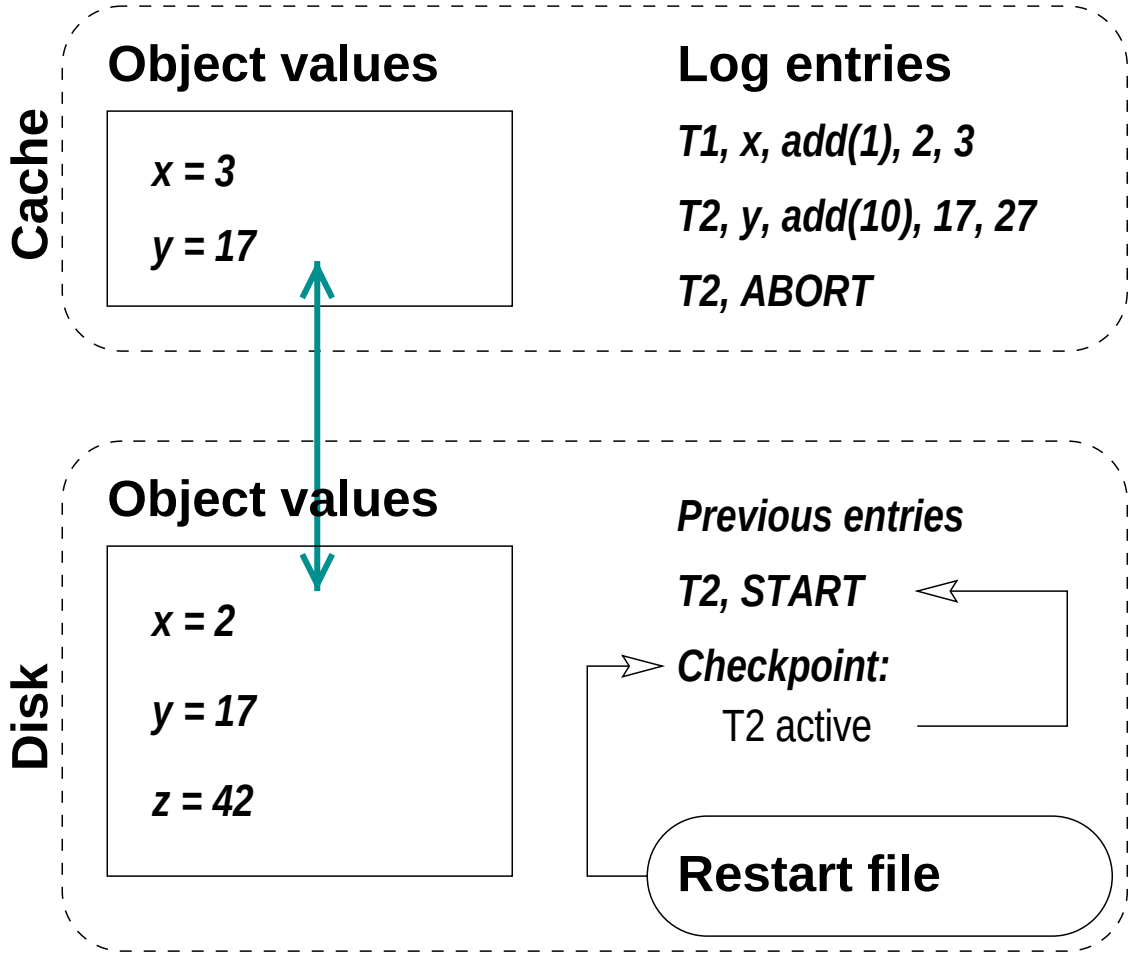
Crash during 1 $\Rightarrow$ no updates performed

Crash during 2 $\Rightarrow$ re-check log, either undo (so no changes) or redo (so all changes made)

Persistent storage – logging (2)

More generally we can record details of multiple transactions in the log by associating each with a *transaction id*. Complete records, held in an append-only log, may be of the form:

- $(transaction, operation, old, new)$
- or $(transaction, \text{start/abort/commit})$



Persistent storage – logging (3)

We can cache values in memory and use the log for recovery

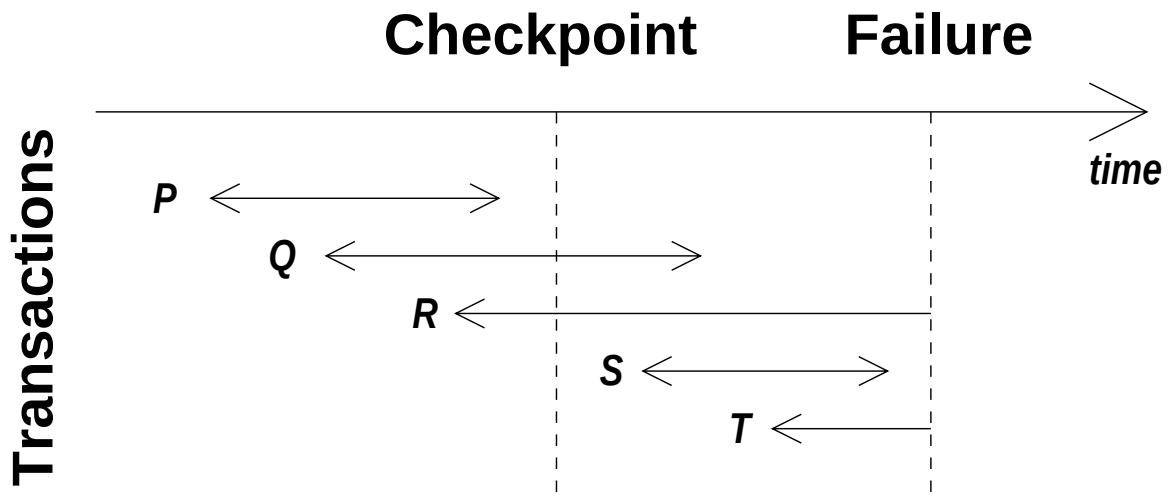
- A portion of the log may also be held in volatile storage, but records for a transaction must be written to non-volatile storage before that transaction commits
- Values can be written out lazily: the system state can be recovered using the log

A naïve implementation would be inefficient, e.g. when aborting a transaction. A checkpoint mechanism can be used, e.g. every x seconds or every y log records. For each checkpoint:

- Force log records out to non-volatile storage
- Write a special *checkpoint record* that identifies the then-active transactions
- Force cached updates out to non-volatile storage

Then write the location of the checkpoint record into a *restart file*

Persistent storage – logging (4)



P already committed before the checkpoint – any items cached in volatile storage must have been flushed

Q active at the checkpoint but subsequently committed – log entries must have been flushed at commit, REDO

R active but not yet committed – UNDO

S not active but has committed – REDO

T not active, not yet committed – UNDO

Persistent storage – logging (5)

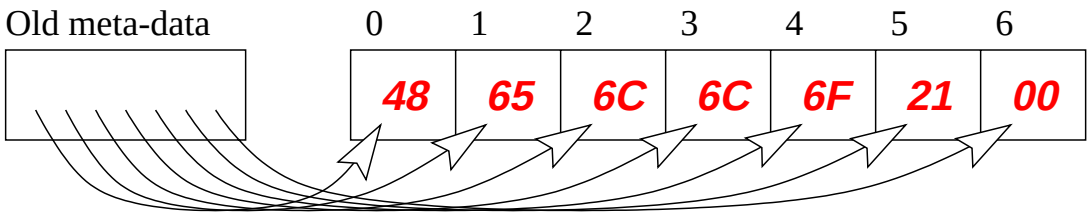
A general algorithm for recovery:

- The *recovery manager* keeps UNDO and REDO lists
- Initialize UNDO with the set of transactions active at the last checkpoint
- REDO is initially empty
- Search forward from the checkpoint record:
  - Add transactions that `start` to the UNDO list
  - Move transactions that `commit` from the UNDO list to the REDO list
- Then work backwards through the log from the end to the checkpoint record:
  - UNDOing the effect of transactions on the UNDO list
- Then work forwards from the log from the checkpoint record:
  - REDOing the effect of transactions in the REDO list

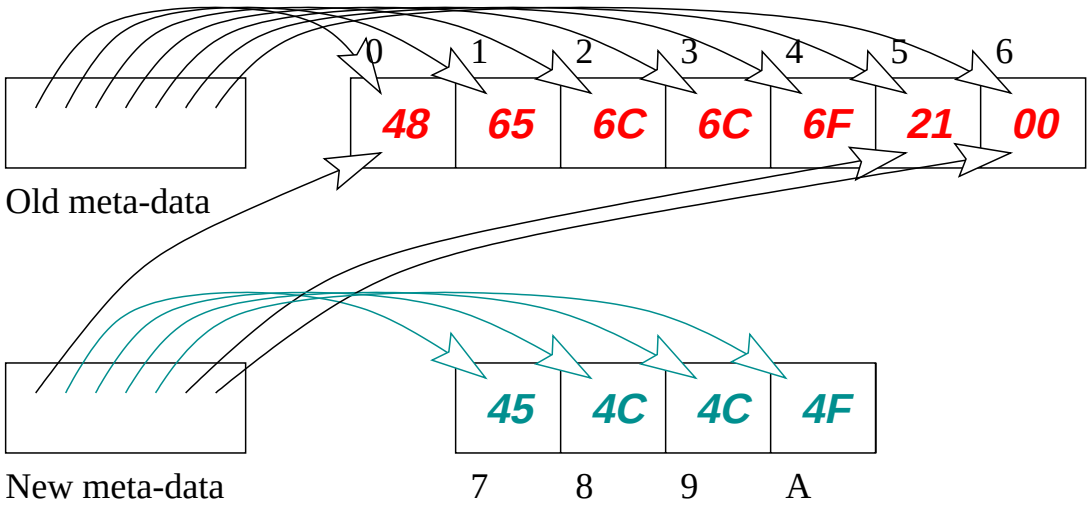
Storing old and new values in the log enables general *idempotent* UNDO and REDO

Persistent storage – shadowing

An alternative to logging: create separate old and new versions of the data structures being changed



An update starts by constructing a new 'shadow' version of the data, possibly sharing unchanged components:



The change is committed by a single in-place update to a location containing a pointer to the current version. This last change must be guaranteed atomic by the system.

How can this be extended for persistent updates to multiple objects?

Isolation

Recall our original example:

```
transactionally {
  if (source.balance() >= amount) {
    source.withdraw (amount);
    destination.deposit (amount);
    return true;
  } else {
    return false;
  }
}
```

What can the system do in order to enforce *isolation* between transactions specified in this manner?

A simple approach: execute transactions serially, allowing only one to operate at a time

✔ Simple, 'clearly correct', independent of the operations performed within the transaction

✘ Does not enable concurrent execution, e.g. two of these operations on separate sets of accounts

✘ What happens if operations can fail?

Isolation – serialisability

This idea of executing transactions serially provides a useful correctness criteria for executing transactions in parallel:

- A concurrent execution is *serialisable* if there is some serial execution of the same transactions that gives the same result

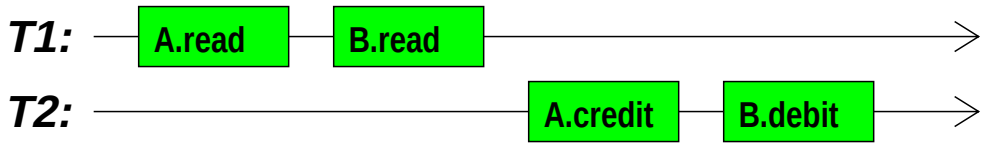
Suppose we have two transactions:

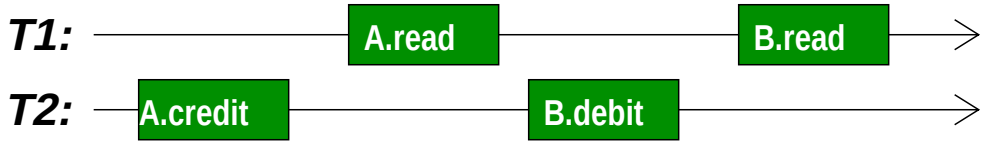
```
T1: transactionally {
  int s = A.read ();
  int t = B.read ();
  return s + t;
}

T2: transactionally {
  A.credit (100);
  B.debit (100);
}
```

If we assume that the individual `read`, `credit` and `debit` operations are implemented atomically (e.g. by `synchronized` methods) then an execution without further concurrency control can proceed in 6 ways

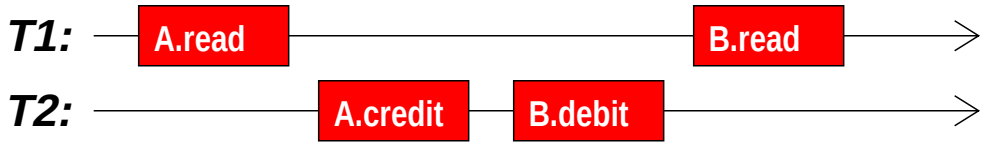
Isolation – serialisability (2)

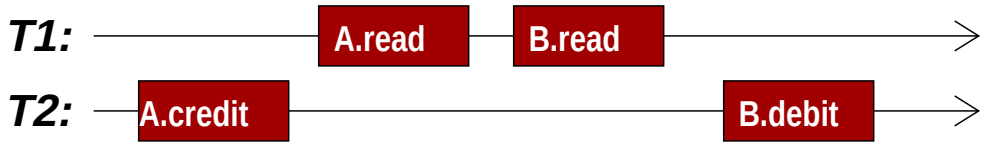
Both of these concurrent executions are OK:





Neither of these concurrent executions is valid:





In each case some – but not all – of the effects of `T2` have been seen by `T1`, meaning that we have not achieved *isolation* between the transactions

Isolation – serialisability (3)

We can depict *a particular execution* of a set of concurrent transactions by a *history graph*

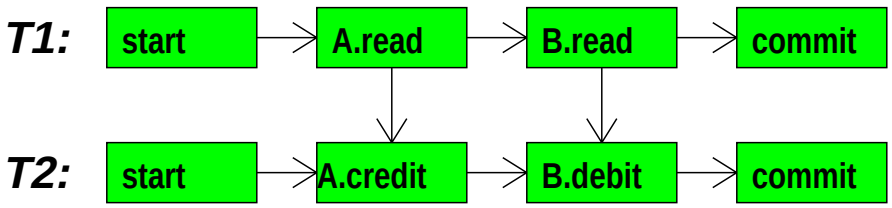
- Nodes in the graph represent the operations comprising each transaction, e.g. `T1: A.read`
- An directed edge from node `a` to node `b` means that `a` *happened before* `b`
  - Operations within a transaction are totally ordered by the *program order* in which they occur
  - Conflicting operations on the same object are ordered by the object's implementation

For clarity we usually omit edges that can be inferred by the transitivity of *happens before*

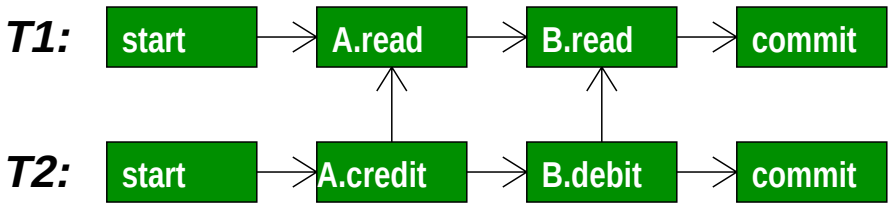
Suppose again that we have two objects `A` and `B` associated with integer values and run transaction `T1` that reads values from both and transaction `T2` that adds to `A` and subtracts from `B`

Isolation – serialisability (4)

These histories are OK. Either both the `read` operations see the old values of A and B:

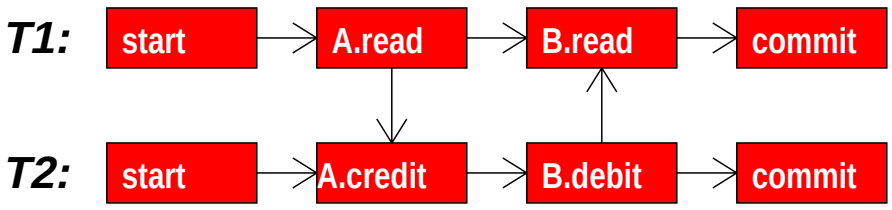


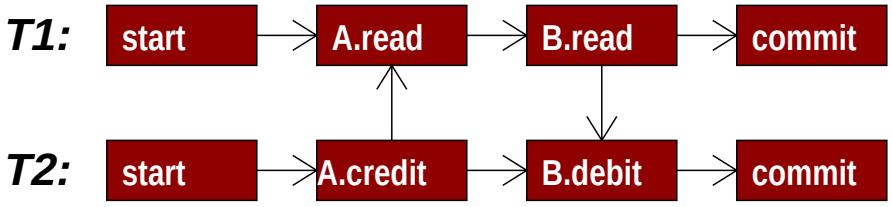
or both `read` operations see the new values:



Isolation – serialisability (5)

These histories show non-serialisable executions in which one `read` sees an old value and the other sees a new value:

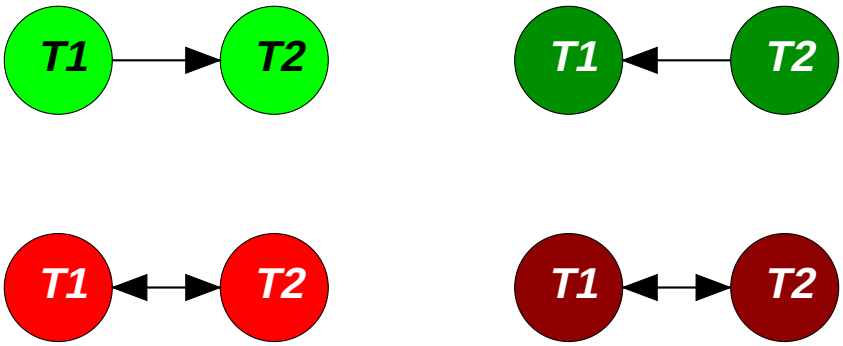




Isolation – serialisability (6)

We can derive a simpler *serialisation graph* in which nodes represent transactions and a directed edge from node T_a to T_b means that some node in T_b's history graph is reachable from some node in T_a's

A history is serialisable iff its serialisation graph is acyclic



These graphs show whether a particular execution of the transactions corresponds to a serialisable execution

As we've seen in this example, one piece of code can lead to both serialisable and non-serialisable histories

The transaction management system is responsible for ensuring that a serialisable execution is chosen at run-time

Isolation – two-phase locking

We'll now look at some mechanisms for ensuring that transactions are executed in a serialisable manner while allowing more concurrency than an actual serial execution would achieve

In *two-phase locking* (2PL) each transaction is divided into

- a phase of acquiring locks
- a phase of releasing locks

Locks must exclude other operations that may conflict with those to be performed by the lock holder. Simple mutual exclusion locks may suffice, but could limit concurrency. In the example we could use a MRSW lock, held in read mode for `read` and write mode for `credit` and `debit`

- If T_a performs an operation that comes before a conflicting one by T_b then T_a must have released a lock on the object and T_b acquired one
- At that point T_a must have entered its releasing phase – it can't acquire locks on further objects that T_b may have previously updated

Isolation – two-phase locking (2)

How does the system know when (and how) to acquire and release locks if transactions are defined in the form:

```
1 transactionally {
2   if (source.balance() >= amount) {
3     source.withdraw (amount);
4     destination.deposit (amount);
5     return true;
6   } else {
7     return false;
8   }
9 }
```

- Could require explicit invocations by the programmer, e.g. additional operations to
  - acquire a read lock on `source` before 2, release if the `else` clause is taken,
  - upgrade to a write lock on `source` before 3,
  - acquire a write lock on `destination` before 4,
  - release the lock on `source` any time after acquiring both locks,
  - release the lock on `destination` after 4

Isolation – two-phase locking (3)

How well would this form of two-phase locking work?

- ✔ Ensures serialisable execution if implemented correctly
- ✔ Allows arbitrary application-specific knowledge to be exploited, e.g. using MRSW for increased concurrency over mutual exclusion locks
- ✔ Allowing other transactions to access objects as soon as they have been unlocked increases concurrency
- ✘ Complexity of programming (e.g. 2PL $\Rightarrow$ MRSW needs an *upgrade* operation here)
- ✘ Risk of deadlock
- ✘ If $T_a \rightarrow T_b$ then isolation requires that
  - T_b cannot commit until T_a has
  - T_b must abort if T_a does ('cascading aborts')

Some of these problems can be addressed by *strict isolation* in which all locks are held until release: transactions *never* see partial updates made by others

With *Strict 2PL* locks are only released when a transaction commits or aborts – no cascading aborts but consider the effect of long transactions...

Isolation – timestamp ordering

Timestamp ordering (TSO) is another mechanism to enforce isolation:

- Each transaction has a timestamp – e.g. of its start time. These must be totally ordered, using a suitable tie-break if necessary
- Each object requires fields to hold
  - The timestamp of the most recent transaction
  - The operation invoked upon it
- Each time an operation is invoked that conflicts with the previous one on the object:
  - ✔ It is allowed to proceed if it is from a transaction with a later timestamp
  - ✘ It is rejected as *too late* if it is from an earlier transaction

Isolation – timestamp ordering (2)

One serialisable order is achieved: that of the timestamps of the transactions, e.g.

```
T1,1: start          T2,1: start
T1,2: A.read()       T2,2: A.credit()
T1,3: B.read()       T2,3: B.debit()
```

✔ `T1,1` executes, → timestamp 17

✔ `T1,2` executes, A: 17,`read`

✔ `T2,1` executes, → timestamp 42

✔ `T2,2` executes, OK (later) A: 42,`credit`

✔ `T2,3` executes, B: 42,`debit`

✘ `T1,3` attempted: too late 17 earlier than 42 and `read` conflicts with `credit`

In this case both transactions could have committed if `T1,3` had been executed before `T2,3`

Isolation – timestamp ordering (3)

✔ The decision of whether to admit a particular operation is based on information local to the object

✔ Simple to implement – e.g. by interposing the checks on each invocation (contrast with 2PL)

✔ Avoiding locking may increase concurrency (but see below: the work performed may not be useful)

✔ Deadlock is not possible

✘ Cascading aborts are possible – e.g. if `T1,2` had updated `A` then it would need to be undone and `T2` would have to abort because it may have been influenced by `T1`

— could delay `T2,2` until `T1` either commits or aborts (still avoiding deadlock)

✘ Serialisable executions can be rejected if they do not agree with the transactions timestamps (e.g. executing `T2` in its entirety, then `T1`)

Generally: the low overheads and simplicity make TSO good when conflicts are rare

Isolation – OCC

Optimistic Concurrency Control (OCC) is another mechanism for enforcing isolation

A transaction operates on *shadow copies* of objects: changes remain local. Copies may be taken at transaction start or perhaps each time it accesses a new object

Upon commit:

- *Validate* that the the shadows were consistent...
- ...and no other transaction has committed an operation on an object which conflicts with one intended by this transaction

✔ If OK then commit the updates to the persistent objects, in the same transaction-order at every object

✘ If not OK then abort: discard shadows and retry

Note that abort is easy: just discard the shadows

No cascading aborts or deadlock

But conflicts force transactions to retry

Isolation – OCC (2)

Validation is the complex part of OCC. As usual there are trade-offs between the implementation complexity, generality and likelihood that a transaction must abort

We'll consider a validation scheme using

- a single-threaded validator
- the usual distinction between *conflicting* and *commutative* operations

Transactions are assigned timestamps when they pass validation, defining the order in which the transactions have been serialised. We'll assign timestamps when validation starts and then either

- confirm during validation that this gives a serialisable order, or
- discover that it does not and abort the transaction

Elaborate schemes are probably unnecessary: OCC assumes transactions do not usually conflict

Isolation – OCC (3)

The validator maintains a *preceding transactions list*:

Validated transaction	Validation timestamp	Objects updated	Committed
P	10	A, B, C	Yes
Q	11	D	Yes
R	12	A, E	

Transactions P and Q have been validated and committed to persistent storage. R has been accepted by the validator but its updates to objects A and E not yet committed

A current timestamp is maintained by each object, holding the validation timestamp of the most recent transaction committed to it:

Object	Timestamp
A	12
B	10
C	10
D	11
E	10

The update to E remains to take place

Isolation – OCC (4)

Before execution:

- Record the validation timestamp of the most recently validated but not committed transaction – in this case 12. This will be the *base timestamp*

Validation phase 1:

- Compare each shadow's timestamp against the base timestamp
  - ✔ Shadow earlier (B,C,D,E): part of a consistent snapshot before 12
  - ✘ Otherwise (A): it may have seen a subsequent update

Validation phase 2:

- Compare the transaction T against each entry (T_{old}) in the list
  - ✔ T_{old} before the base timestamp
  - ✔ T_{old} has no conflicting updates
  - ✘ Otherwise abort T

Isolation – recap

We've seen three schemes:

1. 2PL uses explicit locking to prevent concurrent transactions performing conflicting operations. Strict 2PL enforces strict isolation and avoids cascading aborts. Both may allow deadlock

   ✔ Use when contention is likely and deadlock avoidable. Use strict 2PL if transactions are short or cascading aborts problematic

2. TSO assigns transactions to a serial order at the time they start. Can be modified to enforce strict isolation. Does not deadlock but serialisable executions may be rejected

   ✔ Simple and effective when conflicts are rare. Decisions are made local to each object: suitable for distributed systems

3. OCC allows transactions to proceed in parallel on shadow objects, deferring checks until they try to commit

   ✔ Good when contention is rare. Validator may allow more flexibility than TSO

Example

Finally, we'll look at an example implementing TSO in Java

- This is, of course, only looking at enforcing isolation between transaction – there is no persistent storage
- The syntax is more cumbersome than the

```
transactionally {
    ...
}
```

  notation that may be desired in that the programmer must use a `try...catch` block to deal with aborting transactions and must pass an additional transaction object to each method

- In overview, an instance of `TSOTransaction` is created for each transaction performed. This is used to keep track of the objects that transaction accesses (instantiated from sub-classes of `TSOTransactorObject`). It records the old state of each object to allow transactions to abort

Example – main program

```
import java.util.Random;

class Example {
  static Account a = new Account (100);
  static Account b = new Account (100);
  static volatile int u, a1, a2;

  public static void main (String args[]) {
    Thread t1 = new Thread () {
        public void run () {
          Random r = new Random ();
          while (true) {
            Transaction tx = null;
            try {
              tx = new TSOTransaction ();
              int n = r.nextInt ();
              b.delta (tx, -n);
              a.delta (tx, n);
              tx.commit (); u++;
            } catch (Failure f) {
              tx.abort (); a1++;
            }
          }
        }
      };
```

Example – main program (2)

```
    Thread t2 = new Thread () {
        public void run () {
          while (true) {
            Transaction tx = null;
            int total;
            try {
              tx = new TSOTransaction ();
              total = a.read (tx) + b.read (tx);
              tx.commit ();
              System.out.println (
                  "Total=" + total +
                  " (" + u + "," + a1 +
                  "," + a2 + ")");
            } catch (Failure f) {
              tx.abort (); a2++;
            }
          }
        }
      };

    t1.start ();
    t2.start ();
  }
}
```

Example – Account

```
class Account extends TSOTransactorObject
{
  int value;

  Account (int value) { this.value = value; }

  Object getState () {
    return new Integer (value);
  }

  void setState (Object o) {
    value = ((Integer)o).intValue();
  }

  void delta (Transaction tx, int change)
                throws Failure
  {
    enter (tx); value += change;
  }

  int read (Transaction tx) throws Failure
  {
    enter (tx); return value;
  }
}
```

Example – TSOTransactorObject

```
abstract class TSOTransactorObject  {
  TSOTransaction mostRecent;

  synchronized void enter (Transaction t)
       throws Failure
  {
    TSOTransaction tx = (TSOTransaction) t;

    if (mostRecent != null &&
        mostRecent != tx) {
      if (tx.earlierThan (mostRecent)) {
        throw new Failure ("Too late");
      } else {
        mostRecent.waitFor ();
      }
    }

    mostRecent = tx;
    tx.enter (this, getState ());
  }

  abstract Object getState ();

  abstract void setState (Object o);
}
```

Example – Transaction

```
abstract class Transaction {
  public static int STATUS_ACTIVE = 0;
  public static int STATUS_COMMITTED = 1;
  public static int STATUS_ABORTED = 2;

  int status = STATUS_ACTIVE;

  abstract void abort ();

  abstract void commit () throws Failure;

  synchronized void waitFor () throws Failure {
    try {
      if (status == STATUS_ACTIVE) wait ();
    } catch (InterruptedException ie) {
      throw new Failure("Interrupted");
    }
  }

  synchronized void setStatus (int status) {
    this.status = status;
    notifyAll ();
  }
}
```

Example – TSOTransaction

```
import java.util.Vector;

class TSOTransaction extends Transaction {
  Vector os = new Vector ();
  Vector states = new Vector ();

  long id = getNextId ();

  static long nextId;

  static synchronized long getNextId () {
    return nextId ++;
  }

  boolean earlierThan (TSOTransaction other) {
    return (id < other.id);
  }

  void enter (TSOTransactorObject o,
              Object old) {
    os.addElement (o);
    states.addElement (old);
  }
```

Example – TSOTransaction (2)

```
  void abort () {
    for (int i = os.size() - 1; i >= 0; i --)
      {
        TSOTransactorObject tso;
        tso = (TSOTransactorObject)
                        os.elementAt (i);
        tso.setState (states.elementAt (i));
      }
    setStatus (STATUS_ABORTED);
  }

  void commit () throws Failure {
    if (status != STATUS_ACTIVE)
      throw new Failure ("Cannot commit");
    setStatus (STATUS_COMMITTED);
  }
}

$ javac *.java && java Example

Total=200 (2,1,0)
Total=200 (1003,2,2)
Total=200 (1095,2,2)
Total=200 (1222,3,4)
...
```