Concurrent Systems and Applications

CST Part 1B
Michaelmas 2008

Lectures 4–9
Distributed Systems & Transactions

Steven Hand

steven.hand@cl.cam.ac.uk

Lecture 4: Distributed systems

Previous section

- Recap on Java

Overview of this section

- Distributed systems
- Naming
- Network communication
- Compound operations
- Crash-tolerance

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 (since all parties share a single copy of the data)
- failures are rare and at the granularity of whole processes (e.g. `SIGKILL` by the user)
- OS-level protection is also performed at the granularity of processes – as far as the OS is concerned a process is running on behalf of one user

Communication between processes (2)

Introducing separate address spaces means that data is not directly shared between the threads involved

- Access mechanisms and appropriate protection must be constructed
- 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)

Any communicating components need

- to agree on how to exchange data – usually by the sender marshalling from a local format into an agreed common format and the receiver unmarshalling

  — similar to using the serialization API to read/write an object to a file on disk

- to agree on how to name shared (or shareable) entities

Distributed systems

More generally, four recurring problems emerge when designing distributed systems:

- Components execute in parallel

  — maybe on machines with very different performance

- Communication is not instantaneous

  — and the sender does not know when/if a message is received

- Components (and/or their communication links) may fail independently

  — usually need explicit failure detection and robustness against failed components/links restarting

- Access to a global clock cannot be assumed

  — different components may observe events in a different order

To varying degrees we can provide services to address these problems. Is complete transparency possible?

Distributed systems (2)

Focus here is on basic naming and communication. Other courses cover access control (Part 1B Intro. Security) and algorithms (Part 2: Distributed Systems, Advanced Systems Topics)

We'll look at two different communication mechanisms:

- ➤ 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
- ➤ Low-level communication using network sockets
  - ✔ A 'lowest-common-denominator': the TCP & UDP protocols are available on almost all platforms
  - ✘ Much more for the application programmer to think about; many wheels to re-invent

Interface definition

The provider and user of a network service need to agree on how to access it and what parameters / results it provides. In Java RMI this is done using Java interfaces

- ✔ 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 separate Interface Definition Language (IDL) is used

- ➤ This provides 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)

(An aside: they must also agree on what the service does, but that needs human intervention!)

Interface definition: OMG IDL

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

```
//POS Object IDL example
module POS {
  typedef string Barcode;

  interface InputMedia {
    typedef string OperatorCmd;
    void barcode_input(in Barcode item);
    void keypad_input(in OperatorCmd cmd);
  };
};
```

- 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, `booleans` and `octets`
- Parameter modifiers `in`, `out` and `inout` define the direction in which parameters are copied

Interface definition: 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;
                   →
                    long _get_value();
                    void _set_value(in long v);
```

Interface definition: 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`
`enum`, `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

Interface defintion: .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 these definitions is available at run-time, e.g. to control marshalling

- 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

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. `lumines.cl.cam.ac.uk`) to addresses (`128.232.10.40`) is considered good practice – i.e. using a name service at run-time to resolve names, rather than embedding addresses directly in a program

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”)

In simple cases 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 by separate allocators if they 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 `/home/smh22/` 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. a UNIX process ID on a multi-processor system does not say on which CPU the process should run, or which user created it

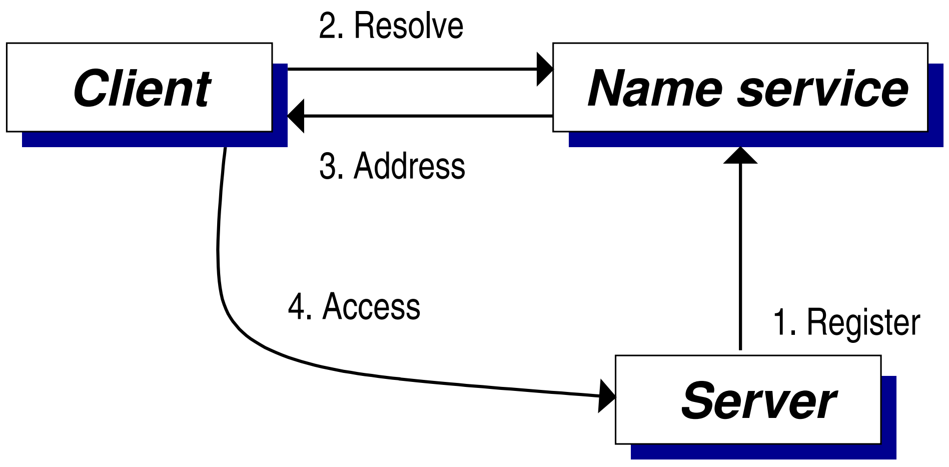
An impure name contains information about the object – e.g. e-mail to `smh22@cam.ac.uk` will always be sent to a mail server in the University

- ➤ Are DNS names, e.g. `lumines.cl.cam.ac.uk` pure or impure?
- ➤ Are IPv4 addresses, e.g. `128.232.10.40` 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

Name services



- 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

How does the client know how to contact 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

Security

In a distributed system, access control is needed to:

- control communication to/from the various components involved,
  - e.g. consider an industrial system with a component on one computer recording the temperature and responding to queries from another computer that controls settings on a machine its attached to
  - how does the controller know that the temperature readings come from the intended probe?
  - how does the probe know that it's being queried by the intended controller?
- control operations that one component does on behalf of users,
  - e.g. a file server may run as the privileged `root` on a UNIX machine
  - when accessing a file on behalf of a remote client it needs to know who that client is and either cause the OS to check access would be OK, or to do those checks itself
- Again, covered more fully in the Part 1B security and Part II distributed systems courses

Security (2)

We'll look at basic sensible things to do when writing distributed systems in Java

- use a security manager class to limit what the JVM is able to do

  — e.g. limiting the IP addresses to which it can connect or whether it is permitted to write to your files

- if using network sockets directly then make the program robust to unexpected input

  — less of a concern in Java than in C...

A 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)

Security (3)

- 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/javase/6/docs/technotes/guides/security/guide/security/index.html`

Exercises

4-1 If you have access both to a big-endian (e.g. SPARC) and a little-endian machine (e.g. Intel) then test whether an object serialized to disk on one is able to be recreated successfully on the other. Examine what happens if the object refers to facilities intrinsic to the originating machine – e.g. if it contains an open `FileOutputStream` or a reference to `System.out`.

4-2 Suppose that two people are communicating by sending and receiving mobile-phone text messages. Messages are delayed by varying amounts. Some messages are lost entirely. Design a way to get reliable communication (so far as is possible). You may need to add information to each message sent, and possibly create further messages in addition to those sent ordinarily.

4-3 Convert the `POS` module definition from OMG IDL into a Java interface that provides similar RMI functionality.

4-4* Suppose that frequent updates are made to part of a hierarchical namespace, while other parts are rarely updated. Lookups are made across the entire namespace. Discuss the use of replication, distribution, caching or other techniques as ways of providing an effective name service.

Lecture 5: Network sockets (TCP & UDP)

Previous lecture

- Distributed systems
- Interface definitions
- Naming

Overview of this lecture

- Communication using network sockets
- UDP
- TCP

Provisions of POSIX 1003.1-2001 (1)

The ‘socket’ system call:

```
int socket(int domain, int type, int protocol);
```

where domain is one of...

```
PF_UNIX, PF_LOCAL   Local communication
PF_INET             IPv4 Internet protocols
PF_INET6            IPv6 Internet protocols
PF_IPX              IPX - Novell protocols
PF_NETLINK          Kernel user interface device
PF_X25              ITU-T X.25 / ISO-8208 protocol
PF_AX25             Amateur radio AX.25 protocol
PF_ATMPVC           Access to raw ATM PVCs
PF_APPLETALK        Appletalk
PF_PACKET           Low level packet interface
```

and type is ...

Provisions of POSIX 1003.1-2001 (2)

SOCK_STREAM
  Provides sequenced, reliable, two-way, connection based byte streams. An out-of-band data transmission mechanism may be supported.

SOCK_DGRAM
  Supports datagrams (connectionless, unreliable messages of a fixed maximum length).

SOCK_SEQPACKET
  Provides a sequenced, reliable, two-way connection-based data transmission path for datagrams of fixed maximum length; a consumer is required to read an entire packet with each read system call.

SOCK_RAW
  Provides raw network protocol access.

SOCK_RDM
  Provides a reliable datagram layer that does not guarantee ordering.

SOCK_PACKET
  Obsolete and should not be used in new programs.

Low-level communication

Two basic network protocols are available in Java:
datagram-based UDP and stream-based TCP

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

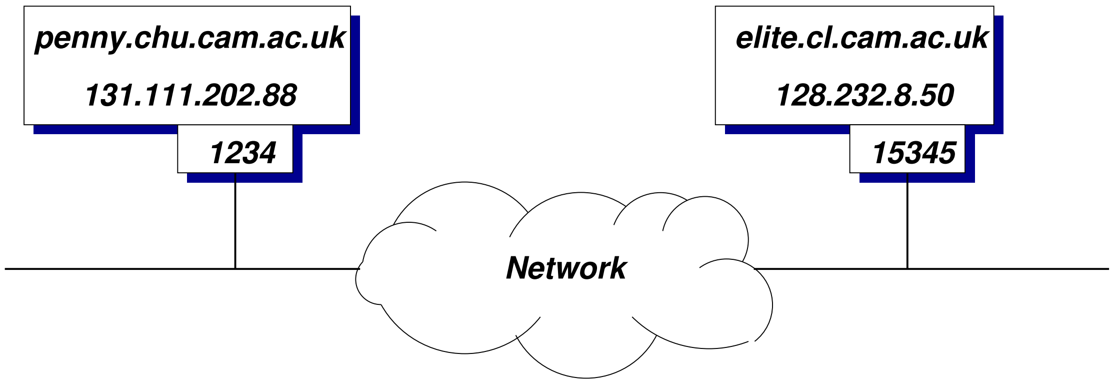
- Loss: datagrams that are sent may never be received,
- Duplication: the same datagram is received several times,
- re-ordering: datagrams are forwarded separately within the network and may arrive out of order

What is provided:

- 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 – a UDP datagram may be fragmented into separate packets within the network, but these are reassembled by the receiver

Low-level communication (2)

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)



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)

As far as we're concerned here, the network acts as a 'magic cloud' that conveys datagrams – see Digital Communication I for layering in general and examples of how UDP is implemented over IP and IP over (e.g.) ethernet

UDP in Java

- 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`
- `MulticastSocket` defines a UDP socket capable of receiving multicast packets. The constructor specifies the port number and then methods

  ```
  joinGroup (InetAddress g);
  leaveGroup (InetAddress g);
  ```

  join and leave a specified group operating on that port
- Multicast group addresses are a designated subset of the IPv4 address space. Allocation policies are still in flux ⇒ check the local policy before using

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 (marshalling)

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 $\Rightarrow$ all machines on the local network – but note problems of port assignment and more generally of multicast group naming)

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
- Marshalling must still be done explicitly – but serialization may help here
- Communication is always 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);
    }
  }
}
```

Server design

The examples have only illustrated the basic use of the operations on `DatagramSocket`, `ServerSocket` and `Socket`:

- ➤ Typically a server would be expected to manage multiple clients

Doing so efficiently can be a problem if there are lots of clients:

- ➤ Could have one thread per client:
  - ✔ Can exploit multi-processor hardware
  - ✘ Many active clients $\Rightarrow$ frequent context switches
  - ✘ The JVM (+ usually the OS) must maintain state for all clients, whether active or not
- ➤ Could have a single thread which services each client in turn:
  - ✔ Simple, avoids context switching
  - ✘ No ‘wait for any input stream’ operation in java (cf `select` in UNIX): must poll each client whether needed or not
- ➤ The `javo.nio` package now supports asynchronous I/O

Exercises

5-1 Write a class `UDPSender` which sends a series of UDP packets to a specified address and port at regular 15 second intervals. Write a corresponding `UDPReceiver` which receives such packets and records the inter-arrival time. How does the performance differ if (i) both programs run on the same computer, (ii) both run on computers on the University network or (iii) one runs on the University network and another on a home or college WiFi internet connection. Do you see packets that are lost, duplicated or re-ordered? Do the packets arrive regularly spaced?

5-2 Write similar classes `TCPSender` and `TCPReceiver` which establish a TCP connection over which single bytes are sent at 15 second intervals. How does the performance compare with the UDP implementation. Is it necessary to call `flush` on the `OutputStream` after sending each byte?

5-3 Consider a server for a noughts-and-crosses game. The two players communicate with it over UDP. Describe a possible structure for the server – in terms of the major data structures, the threads used, the format of the datagrams sent and the concurrency-control techniques.

Lecture 6: RPC & RMI

Previous lecture

- UDP: connectionless, unreliable
- TCP: connection-oriented, reliable

Overview of this lecture

- Java Remote Method Invocation (RMI)

Remote method invocation

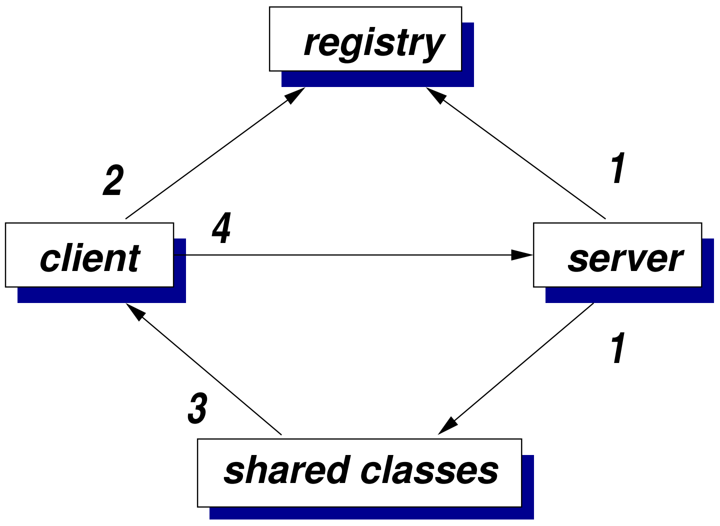
Using UDP or TCP it was necessary to

- 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 in a shared location, the RMI codebase
2. A client queries the registry to obtain a reference to a remote object
3. If they are not directly available, the client obtains the `.class` files needed to access the remote object
4. The client makes an RMI call to the remote object

The registries act as a name service, with names of the form `rmi://linux2.pwf.cl.cam.ac.uk/smh22/myexample`

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 ($\Rightarrow$ 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 RPC systems make different 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—HOWTO

1. Write an interface to define the operations to be exposed by the RMI server.
   - This interface must extend a special interface called `java.rmi.Remote`.
   - All remote methods must throw `RemoteException`.
   - It is common to add a `static final` field to the interface, declaring the `String` name we intend to use with the RMI Registry: putting the name in the shared interface ensures that the client and the server use the same name.
2. The `.class` file generated by compiling the interface should be placed in the RMI codebase so it is available to the server and to the client applications.
3. Write the server program, which will contain a class implementing the interface. This implementation class will exist only at the server end. The RMI mechanism will allow client applications to send requests to objects in our server.
4. The server's main routine should instantiate the implementation class, register it with the RMI Registry using the `String` name from the interface.
   - The server should enter an endless loop, periodically re-registering with the RMI Registry.
5. Write the client…

RMI—Client

- Instead of calling `new` to get an instance of the class which implements our interface, call `Naming.lookup` to get a “stub class” from the RMI Registry.
- The stub is a class generated at runtime by “reflection”; an instance of `java.lang.reflect.Proxy` is instantiated which implements our interface, but in which every method body merely serialises the arguments and sends them over the network to our RMI Server.
- The code requires an exception handler to deal with:
  - `NotBoundException`—no remote object has been associated with the name we tried to look up in the registry
  - `RemoteException`—if the RMI registry could not be contacted (`Naming.lookup`) or if there was a problem with a request to invoke a method).
  - `AccessException`—if the operation has not been permitted by the installed security manager.

RMI—Defining an Interface

Suppose we wish to have a simple "service" which can perform some basic arithmetic:

```
import java.rmi.*;

public interface Calculator extends Remote {
    public static final String NAME
         = "rmi://tempest.cl.cam.ac.uk/smh22/calc";

    public long add(long a, long b)
        throws RemoteException;
    public long sub(long a, long b)
        throws RemoteException;
    public long mul(long a, long b)
        throws RemoteException;
    public long div(long a, long b)
        throws RemoteException;
}
```

- All RMI invocations are made across remote interfaces extending `java.rmi.Remote`
- The field NAME says which RMI registry will be used (the one on `tempest.cl.cam.ac.uk`) and the name to register the service as (`smh22/calc`)
- All remote methods must throw `RemoteException`

RMI—Implementing the Interface

Next write some code which implements the interface; must extend `java.rmi.server.UnicastRemoteObject`.

```
 1 import java.rmi.*;
 2 
 3 public class CalculatorImpl
 4     extends java.rmi.server.UnicastRemoteObject
 5     implements Calculator {
 6 
 7     public CalculatorImpl() throws RemoteException {
 8       super();
 9     }
10 
11     public long add(long a, long b)
12     throws RemoteException {
13       return a + b;
14     }
15 
16     public long sub(long a, long b)
17     throws RemoteException {
18       return a - b;
19     }
20 
21     public long mul(long a, long b)
22     throws RemoteException {
23       return a * b;
24     }
25 
26     public long div(long a, long b)
27     throws RemoteException {
28       return a / b;
29     }
30 }
```

➤ Lines 7–9 are an explicit constructor which we need in order to declare the `RemoteException` exception.

RMI—Writing the Server

The “server” class just creates a new instance of the implementation class and registers the service with the registry:

```
import java.rmi.Naming;

public class CalculatorServer {

  public CalculatorServer() {
    try {

      // Instantiate server
      Calculator c = new CalculatorImpl();

      // Bind name in the registry
      Naming.rebind(Calculator.NAME, c);

      System.out.println(Calculator.NAME +
        " server now available");

    } catch (Exception e) {
      System.out.println("Trouble: " + e);
    }
  }

  public static void main(String args[]) {
    new CalculatorServer();
  }
}
```

- Better in practice to (re)bind the name in the registry periodically (e.g. in case the registry goes away for a while)
- Usually also want a security manager to limit the actions that can be performed.

RMI—Enhancing the Server

```
 1 import java.rmi.Naming;
 2 import java.lang.Thread;
 3
 4 public class CalculatorServer {
 5
 6   public CalculatorServer() {
 7     try {
 8       // Instantiate server
 9       Calculator c = new CalculatorImpl();
10
11       // Install a security manager
12       System.setSecurityManager(
13         new java.rmi.RMISecurityManager());
14
15       while(true) {
16         // Bind name in the registry
17         Naming.rebind(Calculator.NAME, c);
18         System.out.println(Calculator.NAME +
19           " server now available");
20
21         // Repeat every 5 seconds
22         Thread.currentThread().sleep(5000L);
23       }
24
25     } catch (Exception e) {
26       System.out.println("Trouble: " + e);
27     }
28   }
29
30   public static void main(String args[]) {
31     new CalculatorServer();
32   }
33 }
```

- Lines 11–13 install a security manager ⇒ we now require a suitable security policy or the server won't work…

RMI—Security Managers and Security Policies

- By default, applications run without a security manager ⇒ no restrictions are in place
- (Applets, on the other hand, always run with the security manager provided by the web brower)
- To add a security manager to your code, two main options:
  1. Write your own from scratch, or
  2. Use the built-in `java.rmi.RMISecurityManager`
- Option 2 is considerably easier:
  - Default is no access
  - Explicitly grant permissions using a "security policy" file
- E.g. a very permissive `security.policy` file:

```
grant {
  permission java.security.AllPermission;
};
```

- Or a more restrictive one...

```
grant {

  permission java.net.SocketPermission
  "*:1024-65535", "connect,accept";

  permission java.util.PropertyPermission
  "java.rmi.server.codebase", "read";

  permission java.util.PropertyPermission
  "user.name", "read,write";

};
```

RMI—Implementing a Client

```
 1 import java.rmi.Naming;
 2 import java.net.MalformedURLException;
 3 import java.rmi.RemoteException;
 4 import java.rmi.NotBoundException;
 5
 6 public class CalculatorClient {
 7
 8   public static void main(String[] args) {
 9     try {
10       Calculator c = (Calculator)
11          Naming.lookup(Calculator.NAME);
12       System.out.println( c.sub(4, 3) );
13       System.out.println( c.add(4, 5) );
14       System.out.println( c.mul(3, 6) );
15       System.out.println( c.div(9, 3) );
16     }
17     catch (MalformedURLException murle) {
18       System.out.println("MalformedURLException");
19     }
20     catch (RemoteException re) {
21       System.out.println("RemoteException");
22     }
23     catch (NotBoundException nbe) {
24       System.out.println("NotBoundException");
25     }
26     catch (java.lang.ArithmeticException ae) {
27       System.out.println("ArithmeticException");
28     }
29   }
30 }
```

➤ Key thing is lines 9–10: these get you an instance of the stub class implementing the `Calculator` interface.

➤ Invocations on this interface (lines 11–14) are forwarded to the remote object registered as `Calculator.NAME`

Putting it all together

- Select the machine to run the registry and update Calculator.NAME in `Calculator.java`
  - simplest to use same machine for registry & the server
- Compile the interface, implementation, server & client:

```
$ javac Calculator*.java
```

- Generate stub classes for the implementation:

```
$ rmic CalculatorImpl
```

  This produces `CalculatorImpl_Stub.class` and `CalculatorImpl_Skel.class`.
- [Re-]Start the registry:

```
$ rmiregistry &
```

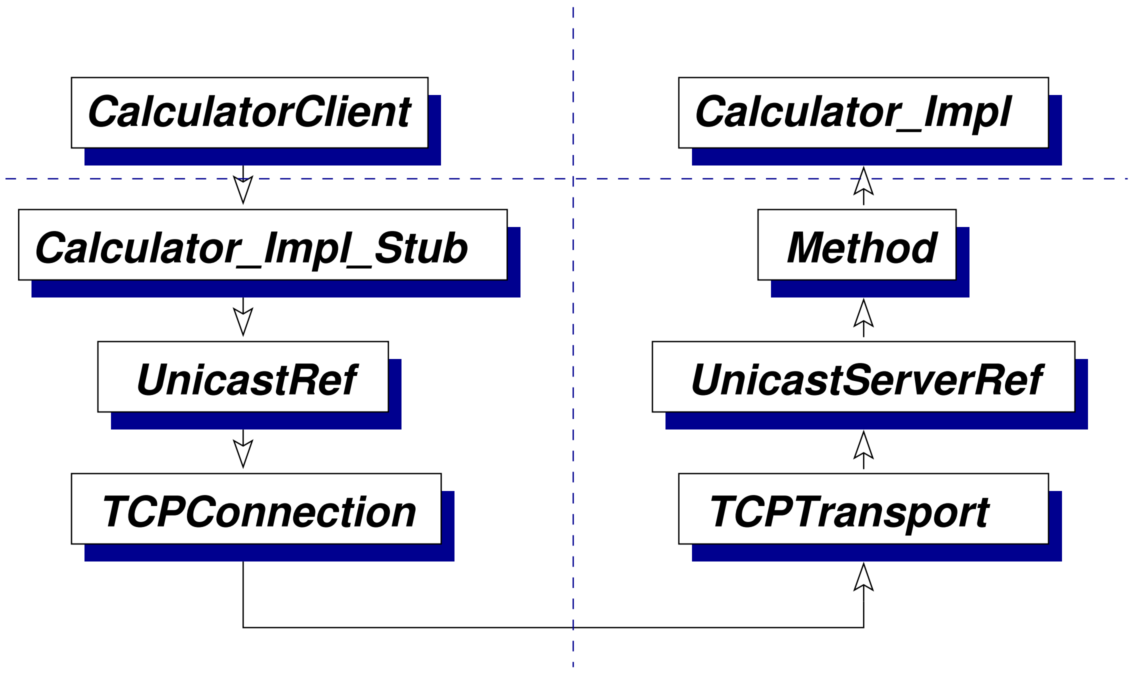
- If you're using the "secure" version of the server:
  - create a `calc.policy` file (see earlier slides)
  - start the server: `$ java -Djava.security.policy=calc.policy CalculatorServer`
- Else just start the server:

```
$ java CalculatorServer
rmi://tempest.cl.cam.ac.uk/smh22/calc server now..
```

- And finally, start the client:

```
$ java CalculatorClient
1
9
18
3
```

RMI implementation



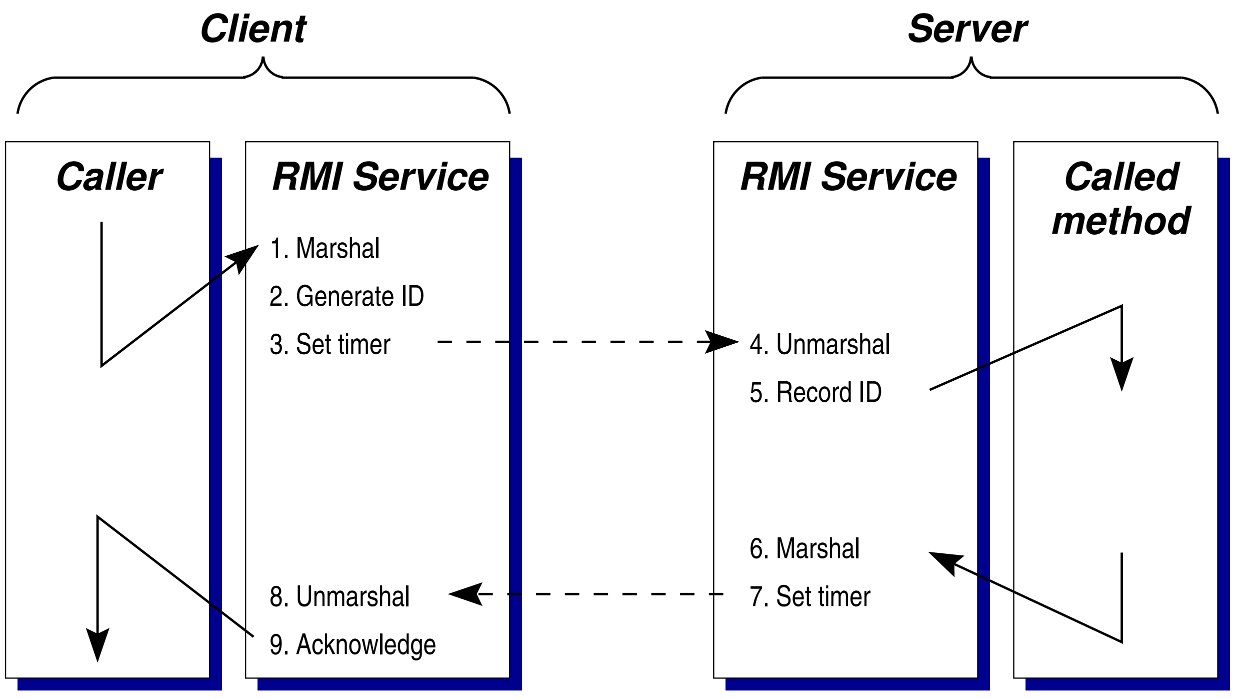
- The `_Stub` class is the one mentioned earlier – it transforms invocations on the `Calculator` interface into generic invocations of an `invoke` method on `UnicastRef`
  - the class may not have quite as human readable a name in modern (1.5+) versions of java
- `UnicastRef` is responsible for selecting 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 object should be prepared to accept concurrent invocations of its methods
- ✔ This concurrency avoids deadlock if remote method `A.m1` invokes an operation on remote method `B.m2` which in turn invokes an operation `A.m3`
- ✘ The application programmer must be aware of how many threads might be created and the impact that they may have on the system
- ➤ Remember: the `synchronized` modifier applies to a method's implementation. It must be applied to the definition in the server class, not the interface

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 a 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

Exercises

6-1 Compile and execute the RMI example yourself.

6-2 Modify the `UDPSender` and `UDPReceiver` example so the sender initiates an RMI call to the receiver at regular 15 second intervals. How does the performance compare now to the UDP and TCP examples?

6-3 To what extent can the fact that a method invocation is remote be made transparent to the programmer? In what ways is complete transparency not possible?

6-4 A client and a server are in frequent communication using the RPC protocol described in the slides and implemented over UDP. Design and outline an alternative protocol that sends fewer datagrams when loss is rare.

6-5* All remote method invocations in Java may throw `RemoteException` because of the failure modes introduced by distribution. Do you agree that `RemoteException` should be a checked exception rather than an unchecked exception (such as `NullPointerException`) which is usually fatal?

Lecture 7: Transactions

Previous section

- Communication using UDP or TCP
- Remote method invocation
- ...and before that concurrency control between threads

Overview of this section

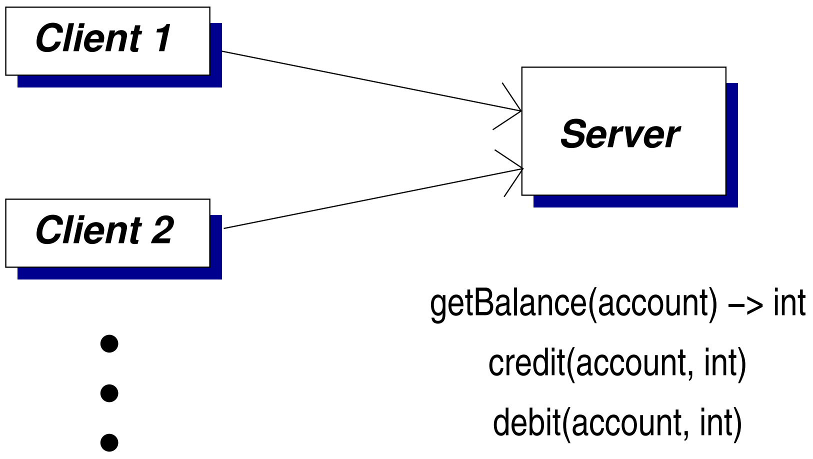
- Compound operations
- Correctness requirements
- Implementation

Compound operations

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.



- Client 1 tries to get the total amount in two accounts
- Client 2 tries to transfer some money between the accounts using `credit` then `debit`
- It can all go horribly wrong, even if `getBalance`, `credit` and `debit` are safe for multi-threaded use

Compound operations (2)

What can go wrong?

- Client 1 may look at the two accounts after one has been credited but before the other is debited
  $\Rightarrow$ the total will be wrong
- Client 2 may crash after doing its `credit` but before the matching `debit`
  $\Rightarrow$ the recipient could be lucky...
- The network may fail, even if the clients are well behaved
- The server may crash

What can be done about these problems?

- Have the server provide `lockAccount` and `unlockAccount` operations
- Have the server provide `transfer` as an atomic operation
- Use some kind of ‘downloadable code’ system

Transactions

Transactions provide a more general abstraction

Ideally the programmer may wish to write something like

```
transaction {
  if (server.getBalance(src) >= amount) {
    server.credit (dest, amount);
    server.debit (src, amount);
    return true;
  } else {
    return false;
  }
}
```

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

- other operations on the same objects
- system crashes (within reason...)

We say that a transaction either commits atomically (if it completes successfully) or it aborts (if it fails for some reason). Aborted transactions leave the state unchanged.

Transactions (2)

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

— the programmer must design transactions that preserve desired invariants, e.g. totals across accounts

Isolation – each committed transaction executes isolated from the concurrent effects of others

— e.g. another transaction shouldn't read the `source` and `destination` amounts mid-transfer and then commit

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 (3)

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

In some cases – where data is held entirely in main memory – we may just be concerned with controlling concurrency

- Note the distinction with the concurrency control schemes based (e.g.) on programmers using mutexes and condition variables: here the system enforces isolation

Isolation

Recall our original example:

```
transaction {
  if (server.getBalance(src) >= amount) {
    server.credit (dest, amount);
    server.debit (src, amount);
    return true;
  } else {
    return false;
  }
}
```

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

A simple approach: have a single lock that's held while executing a transaction, allowing only one to operate at once

- ✔ 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 – serializability

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

- A concurrent execution is serializable if there is some serial execution of the same transactions that gives the same result — the programmer cannot distinguish between parallel execution and the simple one-at-a-time scheme

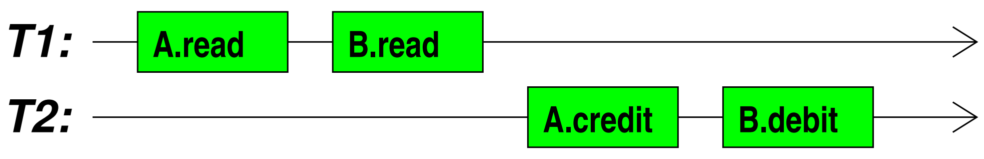
Suppose we have two transactions:

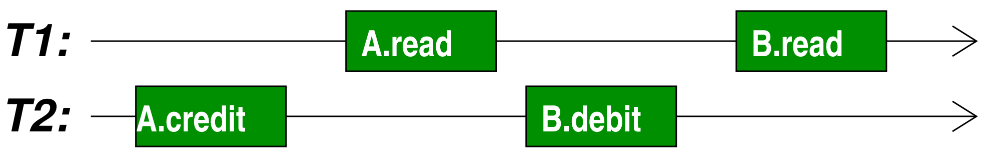
```
T1: transaction {
  int s = server.getBalance (A);
  int t = server.getBalance (B);
  return s + t;
}

T2: transaction {
  server.credit (A, 100);
  server.debit (B, 100);
}
```

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

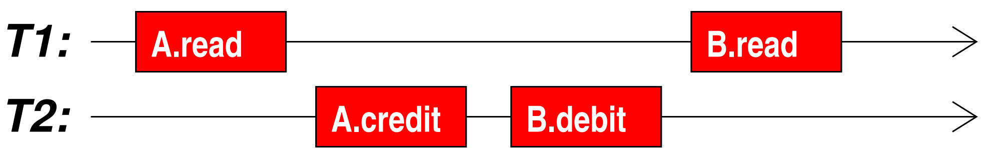
Isolation – serializability (2)

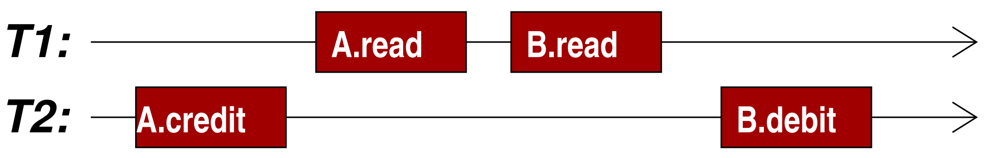
Both of these concurrent executions are OK:





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 – serializability (3)

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

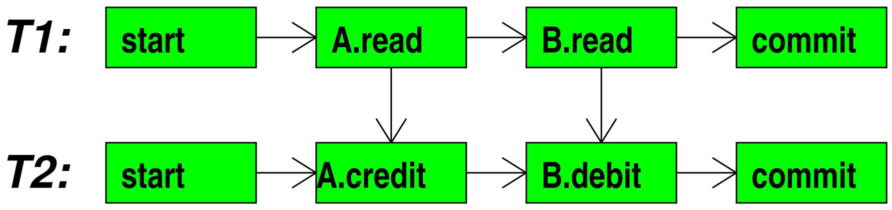
- Nodes in the graph represent the operations comprising each transaction, e.g. `T1:A.read`
- A directed edge from node `a` to node `b` means that `a` happens before `b`
  - Operations within a transaction are totally ordered by the program order in which they occur
  - Conflicting (i.e. non-commutative) 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

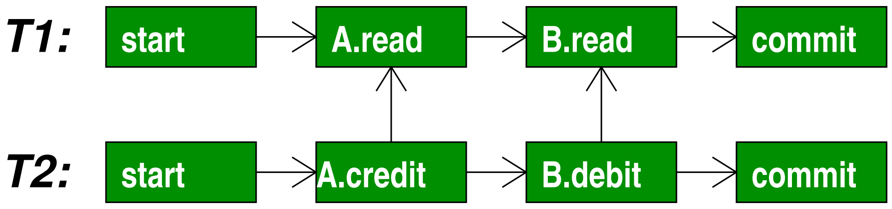
Suppose again that we have two objects `A` and `B` associated with integer values and run transaction `T1` that reads values from both and transaction `T2` that adds to `A` and subtracts from `B`

Isolation – serializability (4)

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

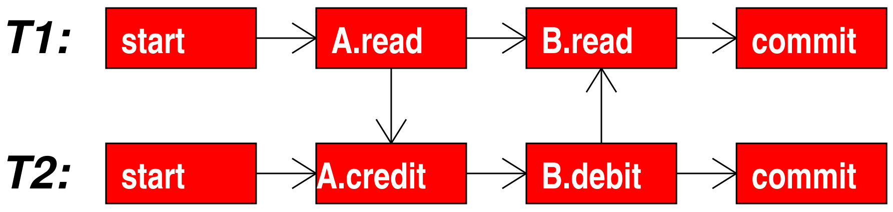


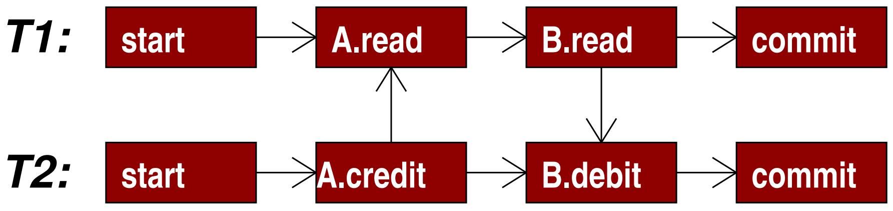
or both `read` operations see the new values:



Isolation – serializability (5)

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





In general, cycles are caused by three kinds of problem:

- Lost updates (e.g. by another transaction overwriting them before they are commmitted)
- Dirty reads (e.g. of updates before they are committed)
- Unrepeatable reads (e.g. before an update by another transaction overwrites it)

Isolation & strict isolation

Here we're interested in avoiding all three kinds of problem so that committed transactions built from simple read and update operations satisfy serializable execution

We can distinguish between enforcing:

- Strict isolation: actually ensure that transactions are isolated during their execution – prohibit all three problems
- Non-strict isolation: ensure that a transaction was isolated before it's allowed to commit

Non-strict isolation may permit more concurrency but can lead to delays on commit (e.g. a transaction that performed a dirty read cannot commit until the writer has) and cascading aborts (if the writer actually aborts)

- NB: in some situations weaker guarantees are accepted for higher concurrency

— In systems using locks to enforce isolation: so long as all transactions avoid lost updates, the decision to avoid dirty & unrepeatable reads can be made on a per-transaction bases

Exercises

7-1 Define the ACID properties for transactions using a simple example (such as transfers between a number of bank accounts) as illustration. For each property, give a possible (incorrect) execution which violates it.

7-2 Slide 7-8 says that there are 6 ways in which execution can proceed, but Slide 7-9 depicts only 4. Illustrate the remaining 2 possible executions and construct history graphs for them.

7-3* If Java were to support a `transaction` keyword then its semantics would need to be defined carefully. Describe how it could behave when the transactional code:

(i) accesses local variables

(ii) accesses fields

(iii) throws exceptions

(iv) makes method calls

(v) uses mutexes and condition variables

(vi) creates threads

You should assume that it is for multi-threaded use on a single computer, rather than needing to support RMI or other kinds of external communication.

Lecture 8: Enforcing isolation

Previous lecture

- Problems of compound operations
- ACID properties for transactions
- Serializability

Overview of this lecture

- Implementing isolation
- Two-phase locking
- Timestamp ordering
- Optimistic concurrency control

Isolation – two-phase locking

We'll now look at some mechanisms for ensuring that transactions are executed in a serializable 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.

Operations can be performed during both phases so long as the appropriate locks are held.

Simple mutual exclusion locks may suffice, but could limit concurrency. In the example we could use a MRSW lock, held in read mode for `getBalance` and write mode for `credit` and `debit`

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 transaction {
2   if (server.getBalance(src) >= amount) {
3     server.credit (dest, amount);
4     server.debit (src, amount);
5     return true;
6   } else {
7     return false;
8   }
9 }
```

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

Isolation – two-phase locking (3)

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

- ✓ Ensures serializable 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)
- ✗ Would be nice to provide `startTransaction` & `endTransaction` rather than individual lock operations
- ✗ Risk of deadlock
- ✗ If T_b locks an object just released by T_a 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 2PL in which all locks are held until commit/abort: transactions never see partial updates made by others

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
- ➤ The ordering between these timestamps will give a serializable order for the transactions
- ➤ If T_a and T_b both access some object then they must do so according to the ordering of their timestamps

Basic implementation:

- ➤ Augment each object with a field holding the timestamp of the transaction that most recently invoked an operation on it
- ➤ Check the object's timestamp against the transaction's each time an operation is invoked:
  - ✔ The operation is allowed if the transaction's timestamp is latest
  - ✘ The operation is rejected as too late if the transaction's timestamp is earlier

Isolation – timestamp ordering (2)

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

```
T1,1:startTransaction      T2,1:startTransaction
T1,2:server.getBalance(A) T2,2:server.credit(A, 100)
T1,3:server.getBalance(B) T2,3:server.debit(B, 100)
```

- ✔ `T1,1` executes, $\longrightarrow$ timestamp 17
- ✔ `T1,2` executes, A: 17,`read`
- ✔ `T2,1` executes, $\longrightarrow$ 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 at the server (contrast with non-strict 2PL)
- ✔ Avoiding locking may increase concurrency
- ✔ Deadlock is not possible
- ✘ Needs a roll-back mechanism
- ✘ 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)
- ✘ Serializable 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 the third kind of mechanism we will look at for enforcing isolation

- Optimistic schemes assume that concurrent transactions rarely conflict
- Rather than ensuring isolation during execution a transaction proceeds directly and serializability is checked at commit time
- Assuming this check usually succeeds (and is itself fast) then OCC will perform well
- ...if the check often fails then performance may be poor because the work done executing the transaction is wasted

For instance consider implementing a shared counter using atomic compare and swap:

```
do {
  old_val = counter;
  new_val = old_val + 1;
} while (CAS (&counter, old_val -> new_val));
```

Isolation – OCC (2)

More generally, a transaction proceeds by taking shadow copies of each object it uses (when it accesses it for the first time). It works on these shadows so changes remain local.

Upon commit it must:

- ➤ 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

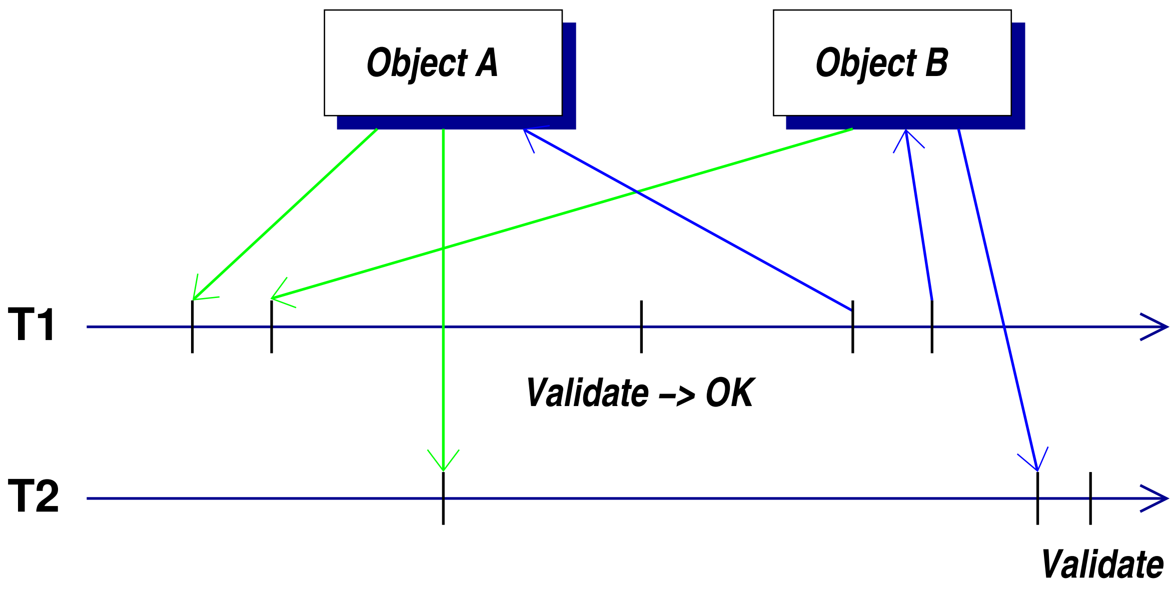
Until commit, updates are made locally. Abort is easy. No need for a roll-back mechanism

No cascading aborts or deadlock

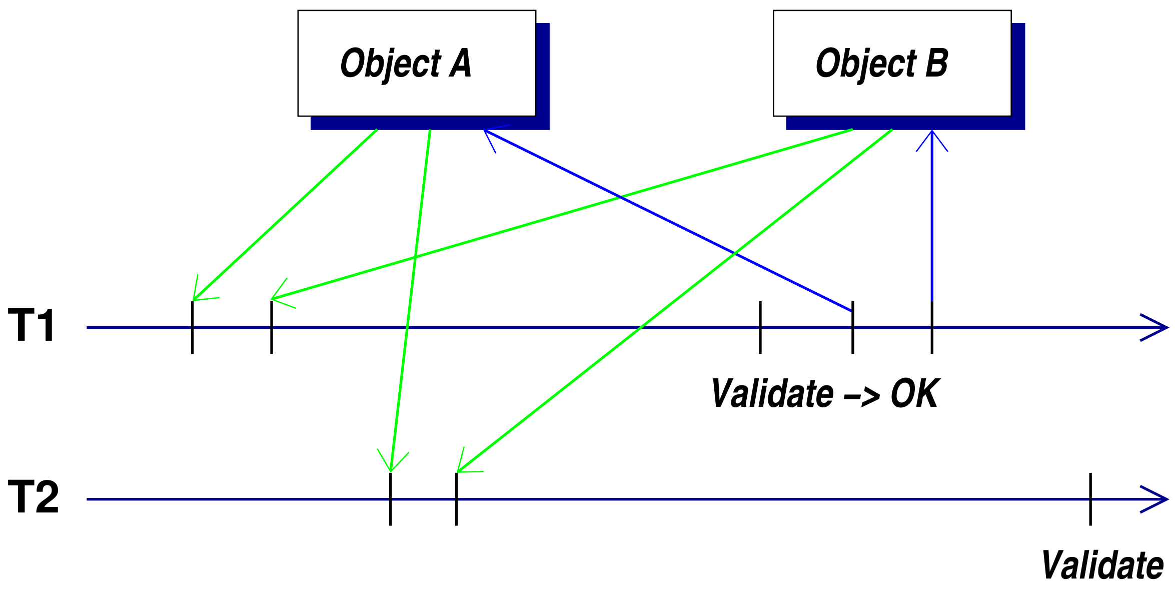
But conflicts force transactions to retry

Isolation – OCC (3)

- The first step avoids unrepeatable reads, e.g. T2 has seen T1's update to B but not seen T1's update to A:



- The second step avoids lost updates, e.g. T1 updates A and B, which T2 would overwrite if it was accepted:



Implementing validation

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 serialized. We'll assign timestamps when validation starts and then either

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

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

Implementing validation (2)

The validator maintains a list of transactions it has accepted:

Validated transaction	Validation timestamp	Objects updated	Updates written back
$T1$	10	A, B, C	Yes
$T2$	11	D	Yes
$T3$	12	A, E	No

- Once a transaction passes validation, it can proceed to write its updates back to the shared objects
- Then the 'written back' flag can be set for it

Each object records the timestamp of the most recent transaction to update it:

Object	Timestamp
A	12
B	10
C	10
D	11
E	9

- In this case $T3$ is still writing back its updates: it's done A but not yet reached E

Implementing validation (3)

Consider $T4$ which updates B and E. Before it starts:

- Record the timestamp of the most recently validated fully-written-back transaction – in this case 11. This will be $T4$'s start time

When $T4$ accesses any object for the first time:

- Take a shadow copy of the object's current state
- Record the timestamp seen (e.g. 10 for B and 9 for E)

Validation phase 1:

- Compare each shadow's timestamp against the start time
  - ✓ Shadow earlier/equal: part of a consistent snapshot at the start time (B, E both OK here)
  - ✗ Shadow later: it may have seen a subsequent update not seen by other shadows

Validation phase 2:

- Compare the transaction $T4$ against each entry in the list after its start time:
  - ✓ No problem if they do not conflict
  - ✗ Abort $T4$ if a conflict is found (with $T3$ on E in this case)

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 serializable executions may be rejected

   ✔ Simple and effective when conflicts are rare. Decisions are made local to each object: well suited 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

Exercises

8-1 A system is to support abortable transactions that operate on a data structure held only in main memory.

(a) Define and distinguish the properties of isolation and strict isolation.

(b) Describe strict two-phase locking (S-2PL) and how it enforces strict isolation.

(c) What impact would be made by changing from S-2PL to ordinary 2PL?

You should say what the consequences are (i) during a transaction's execution, (ii) when a transaction attempts to commit and (iii) when a transaction aborts?

Exercises (2)

8-2 You discover that a system does not perform as well as intended using S-2PL (measured in terms of the mean number of transactions that commit each second). Suggest why this may be in the following situations and describe an enhancement or alternative mechanism for concurrency control for each:

(a) The workload generates frequent contention for locks. The commit rate sometimes drops to (and then remains at) zero.

(b) Some transactions update several objects, then perform private computation for a long period of time before making one final update.

(c) Contention is extremely rare.

Exercises (3)

8-3* A system is using S-2PL to ensure the serializable execution of a group of transactions. Suppose that a new kind of transaction is to be supported which is tolerant to dirty reads and to unrepeatable reads.

(a) Describe how the new transaction could proceed, in terms of when it must acquire and release locks on the objects from which it (i) reads and (ii) updates.

(b) Does supporting this new kind of transaction have any impact on the S-2PL algorithm used by the existing ones?

Past exam questions: 1994 Paper 6 Q7, 2001 Paper 3 Q1

Lecture 9: Crash recovery & logging

Previous lecture

- Enforcing isolation
- Two-phase locking
- Timestamp ordering
- Optimistic concurrency control

Overview of this lecture

- Logging
- Crash recovery
- Checkpoints

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)

If we want the state of an object to be preserved across a machine crash 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

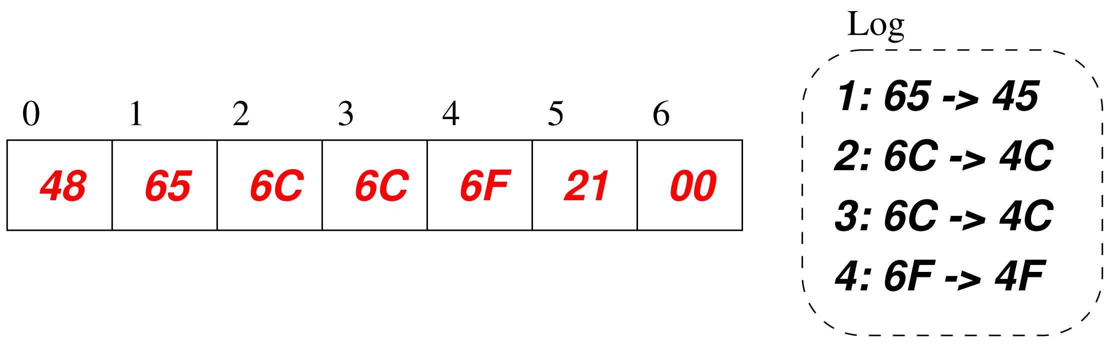
Can we just write object state to disk before every commit? (e.g. invoking `flush()` on any kind of Java `OutputStream`)

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

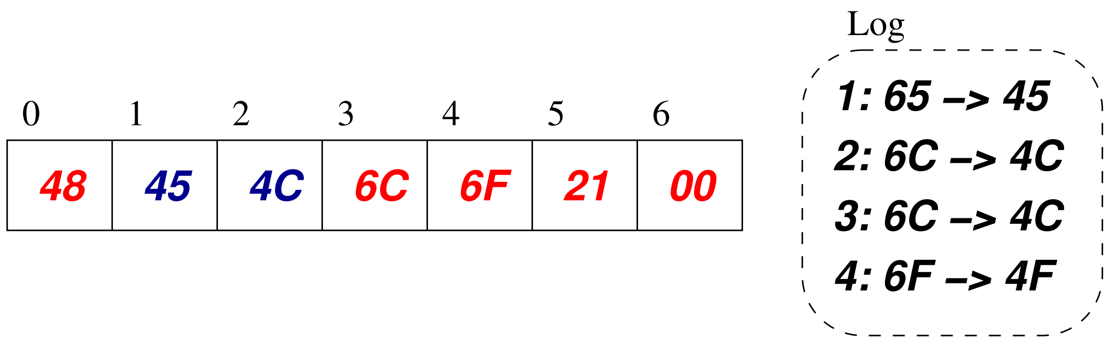
Persistent storage – logging

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



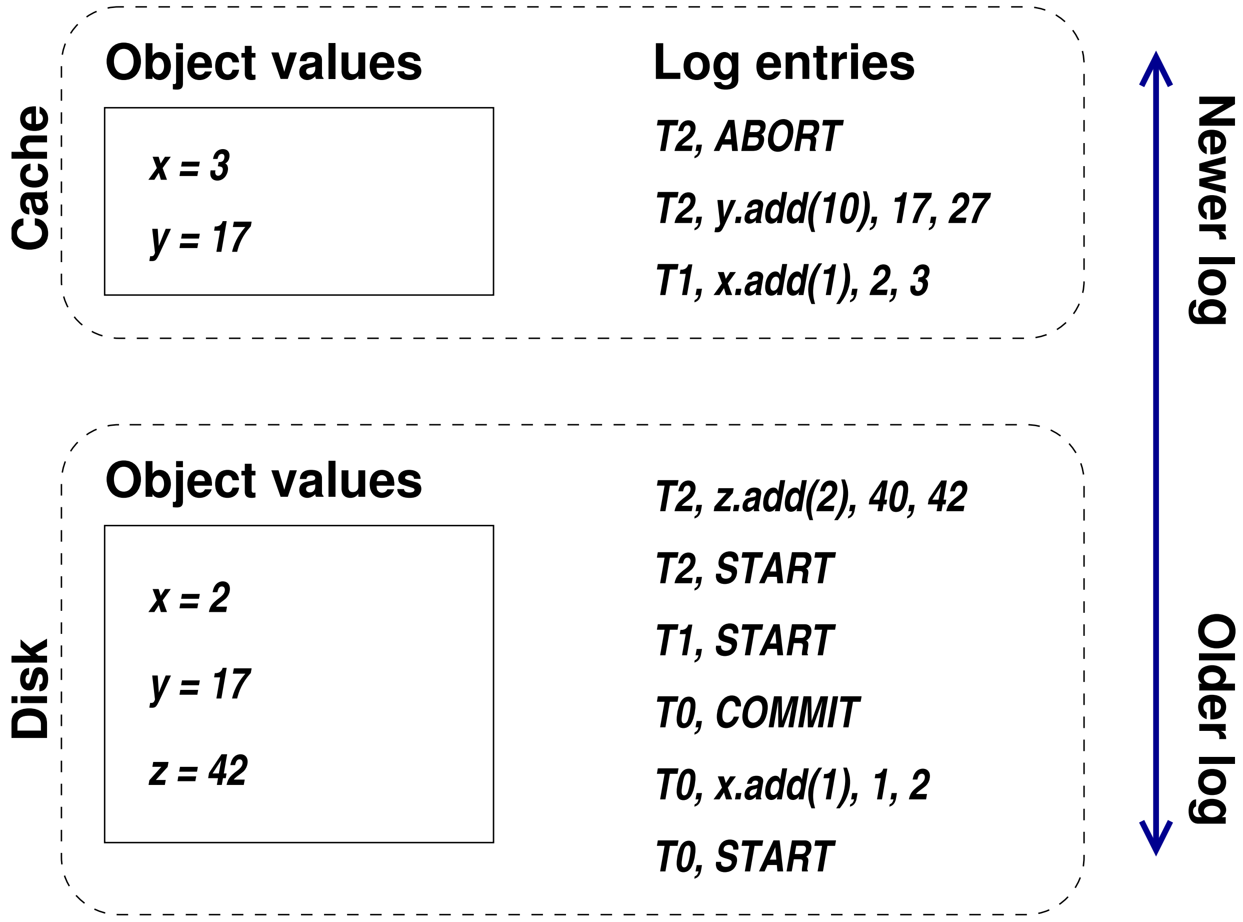
Crash during 1 $\Rightarrow$ no updates performed

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 $(transaction,$ 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

This allows a basic recovery scheme by processing log entries in turn (oldest $\longrightarrow$ youngest)

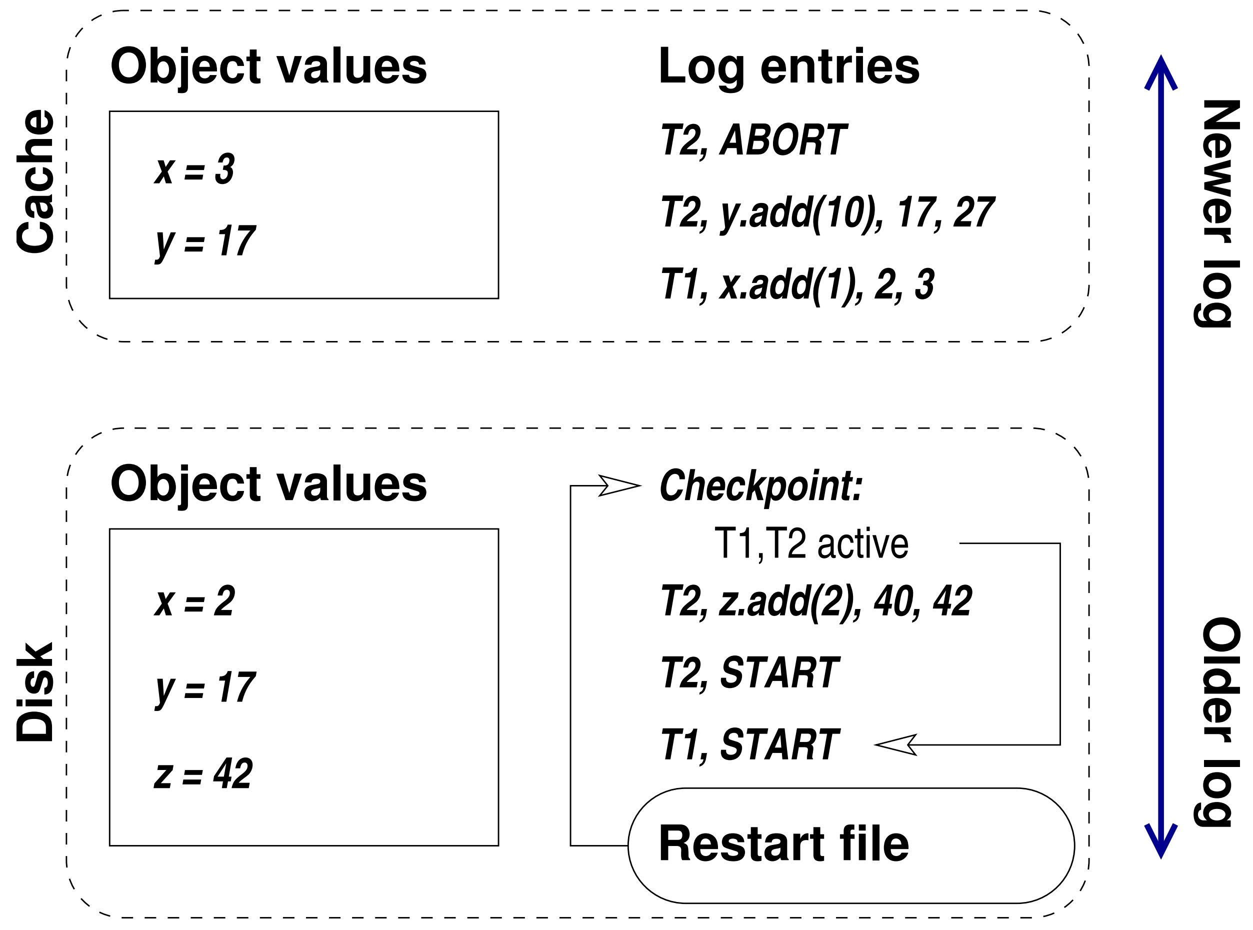
- Note the need for an idempotent record of an update – e.g. for `add` we keep the new & old values as well as the difference
- The old value lets us undo a transaction that's either logged as aborted...
- ...or for which the log stops before we know its outcome

The naïve recovery algorithm can be inefficient

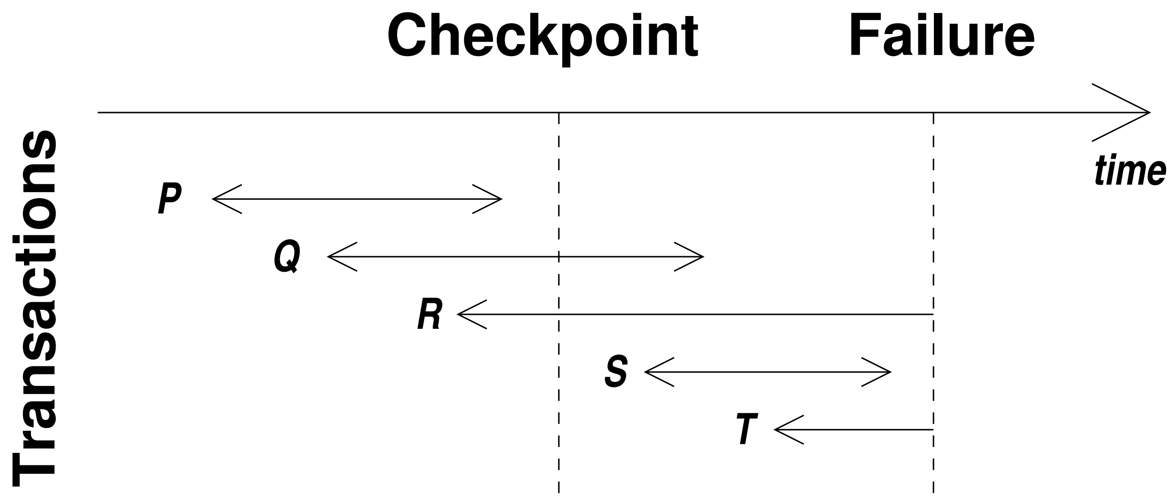
Persistent storage – logging (4)

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



Persistent storage – logging (5)



- 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

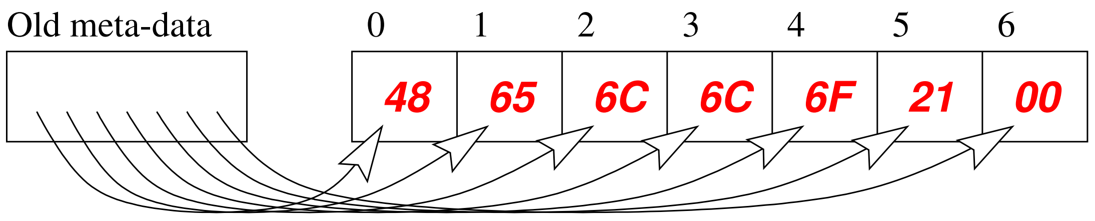
Persistent storage – logging (6)

A general algorithm for recovery:

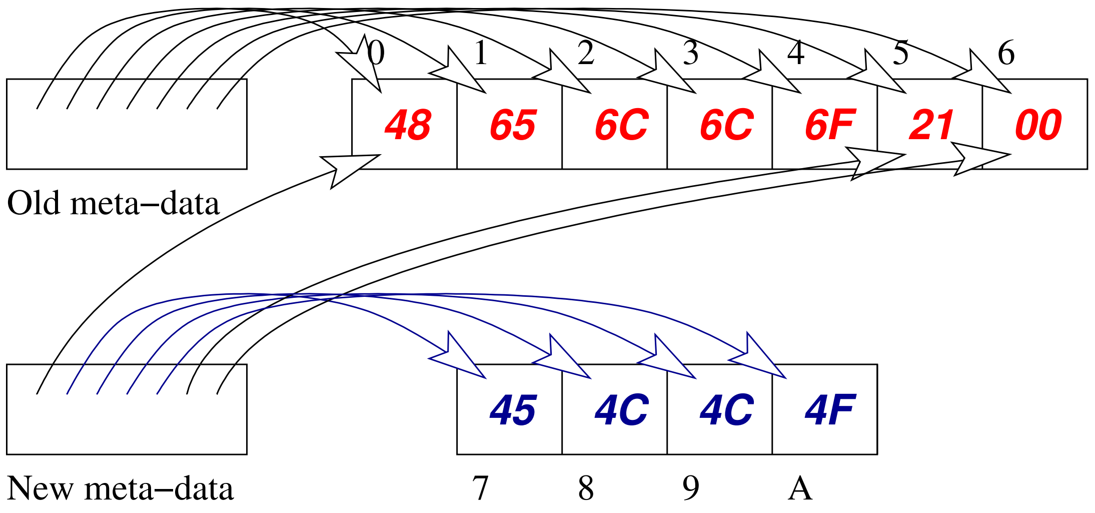
- The recovery manager keeps UNDO and REDO lists
- Initialize UNDO with the set of transactions active at the last checkpoint
- REDO is initially empty
- Search forward from the checkpoint record:
  - Add transactions that `start` to the UNDO list
  - Move transactions that `commit` from the UNDO list 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

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?

Exercises

9-1 Consider the basic logging algorithm (without checkpointing). Show how it enforces atomicity and durability of committed transactions.

While it is not necessary to construct a formal proof, you should be methodical and consider the different operations that the system may perform (e.g. updating objects in memory, starting and concluding transactions, transfers between disk and the in-memory object cache and writing of log entries). Consider the effect of failure and recovery after each one.

9-2 Suppose that you wish to augment the Slime Volleyball game with a high-score table held on disk. Is it necessary to use any of the schemes presented here for persistent storage? If so then suggest which would be most appropriate. If not then say why none is needed.

Past exam questions: 1999 Paper 4 Q2, 1995 Paper 3 Q1, 1997 Paper 4 Q2