UTA

Today's Agenda

- Quick Review
- Continue on Message Passing

UTA

Quick Review

- What is asynchronous/synchronous communication?
- What is link/port/mailbox?

UTA

Message Passing

- Introduction
- Comm. Channels
- Rendzvous and Selective Waits
- Logical Timestamps
- Message-Based Solutions

UTA

What is it?

Message passing is a method for threads to communicate/synchronize with each other by sending and receiving messages.

In a distributed environment, message passing is often the only option that is available for thread communication.

UTA

Asyn. vs. Syn.

Message passing can be asynchronous or synchronous, depending on the types of send and receive operations used.

Asynchronous communication uses non-blocking send and blocking receive. With synchronous communication, both send and receive are blocking.

UTA

Fully Asyn., FIFO and Causal

For asyn. comm., different comm. schemes can be further identified:

- Fully Asyn.: Messages are not necessarily received in the same order as they are sent.
- FIFO: Messages sent from the same sender to the same receiver must be received in the same order as they are sent.
- Causal: Messages sent to the same destination must be received in the same order as they are sent.

UTA

Channel

A communication path between threads is often referred to as a channel, which can be a mailbox, port or link.

- mailbox: many senders and many receivers may access a mailbox object.
- port: many senders but only one receiver may access a port object.
- link: only one sender and one receiver may access a link object.

UTA

Channel (cont'd)

If shared memory is available, then a channel can be implemented as a shared object.

Otherwise, a channel needs to be built on top of a network connection such that messages can be transported across the network.

UTA

Logical Timestamps

Many distributed solutions require the ability to determine the order of events, i.e., which one happens first.

Because of the problem of clock synchronization, events are often stamped with logical time, instead of wall clock time.

Common types of timestamps include integer timestamp, vector timestamp and matrix timestamp.

UTA

Message Passing

- Introduction
- Comm. Channels
- Rendzvous and Selective Waits
- Logical Timestamps
- Message-Based Solutions

UTA

OO Definition

```
public abstract class Channel {
    public abstract void send (Object m);
    public abstract void send ();
    public abstract Object receive ();
}
```


UTA

class Mailbox (syn.)

```
public class Mailbox extends Channel {
    private Object msg = null;
    private final Object sending = new Object ();
    private final Object receiving = new Object ();
    private final BinarySemaphore sent = new BinarySemaphore (0);
    private final BinarySemaphore received = new BinarySemaphore (0);

    public final void send (Object msg) {
        if (msg == null)
            throw new NullPointerException ("Null message passed to send ()"); }
        synchronized (sending) {
            this.msg = msg;
            sent.V ();
            received.P ();
        }
    }
    public final void send () {
        synchronized (sending) {
            msg = new Object ();
            sent.V ();
            received.P ();
        }
    }
    public final Object receive () {
        Object rval = null;
        synchronized (receiving) {
            sent.P ();
            rval = msg;
            received.V ();
        }
        return rval;
    }
}
```

UTA

class Port (syn.)

The send methods for classes Port and Mailbox are the same. As only one thread can receive from a port, the receive method is modified as follows.

```
public final Object receive () {
    synchronized (receiving) {
        if (receiver == null) {
            receiver = Thread.currentThread ();
        }
        if (Thread.currentThread () != receiver) {
            throw new InvalidPortUsage ("Attempted to use port with multiple receivers");
        }

        Object rval = null;
        sent.P ();
        rval = msg;
        received.V ();
    }
    return rval;
}
```


UTA

class Link (syn.)

The receive methods for class Link and Port are the same. As only one sender can send to a link object, the send method needs to check for multiple senders.

UTA

class Mailbox (asyn.)

```
public class Mailbox extends Channel {
    private final int capacity = 100;
    private Object msgs [] = new Object [capacity];
    private CountingSemaphore msgAvail = new CountingSemaphore (0);
    private CountingSemaphore slotAvail = new CountingSemaphore (capacity);
    private final BinarySemaphore senderMutex = new BinarySemaphore (1);
    private final BinarySemaphore receiverMutex = new BinarySemaphore (1);
    private int in = 0, out = 0;

    public final void send (Object msg) {
        if (msg == null)
            throw new NullPointerException ("Null message passed to send ()"); }
        slotAvail.P ();
        senderMutex.P ();
        msgs [in]  = msg;
        in = (in + 1) % capacity;
        senderMutex.V ();
        msgAvail.V ();
        }
    }
    public final void send () {
        // same as send(Object msg) except that "msgs[in] = msg;" becomes "msgs[in] = new Object ();
    }
    public final Object receive () {
        msgAvail.P ();
        receiverMutex.P ();
        Object rval = msgs [out];
        out = (out + 1) % capacity;
        receiverMutex.V ();
        slotAvail.V ();
        return rval;
    }
}
```


UTA

class Port and Link (asyn.)

In the Port class, method receive must check for multiple receivers. In addition, receiverMutex is not needed. (Why?)

In the Link class, methods send and receive must check for multiple senders and receivers. In addition, both senderMutex and receiverMutex are not needed. (Why?)

UTA

Revisit BB problem

```
public final class BoundedBuffer {
    public static void main (String args []) {
        Link deposit = new Link ();
        Link withdraw = new Link ();
        Producer producer = new Producer (deposit);
        Consumer consumer = new Consumer (withdraw);
        Buffer buffer = new Buffer (deposit, withdraw);
        buffer.setDaemon (true);
        buffer.start ();
        producer.start ();
        consumer.start ();
    }
}

final class Message {
    public int number;
    Message (int number) { this.number = number; }
}

final class Producer extends Thread {
    private Link deposit;
    public Producer (Link deposit) {
        this.deposit = deposit;
    }
    public void run () {
        for (int i = 0; i < 3; i ++) {
            System.out.println ("Produced " + i);
            deposit.send(new Message(i));
        }
    }
}
```

```
final class Consumer extends Thread {
    private Link withdraw;
    public Consumer (Link withdraw) {
        this.withdraw = withdraw;
    }
    public void run () {
        for (int i = 0; i < 3; i ++) {
            Message m = (Message) withdraw.receive();
            System.out.println("Consumed " + m.number);
        }
    }
}

final class Buffer extends Thread {
    private Link deposit, withdraw;
    public Buffer (Link deposit, Link withdraw) {
        this.deposit = deposit;
        this.withdraw = withdraw;
    }
    public void run () {
        while (true) {
            Message m = (Message) deposit.receive ();
            withdraw.send(m)
        }
    }
}
```


UTA

Sockets

In a distributed environment, channels are often implemented with support from the kernel.

When two threads want to exchange messages over a channel, each thread creates an endpoint object, which are called sockets.

UTA

Sockets (cont'd)

A sender's socket specifies a local port, and the remote host address and port.

A receiver's socket specifies a local port. Message can be received from any sender.

UTA

TCP vs. UDP

TCP ensures the reliable transmission of messages, meaning that messages will not get lost or corrupted and will be delivered in the correct order.

UDP is a fast but unreliable method of transporting messages. Messages sent using UDP will not be corrupted, but they may be lost of duplicated, or they may arrive out of order.

UTA

TCP sockets in Java (client)

```
// 1. Establish a connection
InetAddress host;
int serverPort = 2020;
Socket socket;
try {
    host = InetAddress.getByName ("www.cs.gmu.edu")
}
catch (UnknownHostException) {
}
try {
    socket = new Socket (host, serverPort);
}
catch (IOException) {
}

// 2. Read/Write data
PrintWriter toServer = new PrintWriter (socket.getOutputStream ());
BufferedReader fromServer = new BufferedReader (new InputStreamReader
(socket.getInputStream));
toServer.println("Hello");
String line = fromServer.readLine ();
System.out.println("Client received: " + line + " from Server");
socket.close ();
```

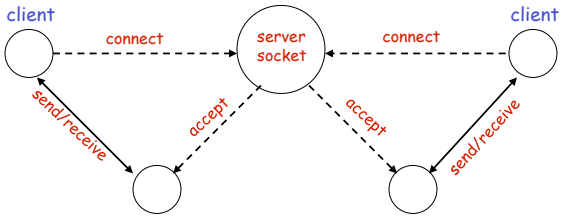

UTA

TCP Sockets in Java (server)

```
int serverPort = 2020;
ServerSocket listen;
try {
    listen = new ServerSocket (serverPort);
    while (true) {
        Socket socket = listen.accept ();
        toClient = new PrintWriter (socket.getOutputStream(), true);
        fromClient =
            new BufferedReader (new InputStreamReader (socket.getInputStream());
        String line = fromClient.readLine ();
        System.put.println("Server received " + line);
        toClient.println("Good-bye");
    }
}
catch (IOException ex) {
}
finally {
}
```

UTA

TCP Comm. Model



UTA

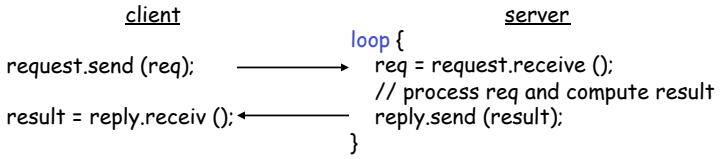
Message Passing

- Introduction
- Comm. Channels
- Rendezvous and Selective Waits
- Logical Timestamps
- Message-Based Solutions

UTA

request/reply

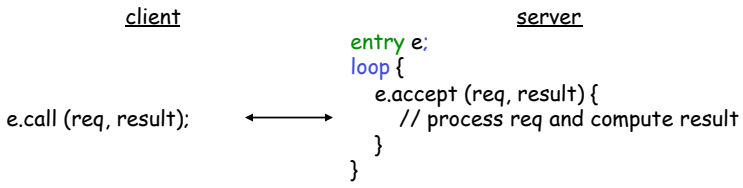
In a client-server environment, the following message-passing paradigm is often involved:



UTA

entry

entry is a new type of channel that facilitates client/ server communication.



UTA

Rendezvous

If no entry call has arrived, the server waits (i.e. blocking).

If one or more entry calls have arrived, the server accepts one call and executes the body of the accept statement.

When the execution of the accept statement is complete, the entry call returns to the client with the server's reply.

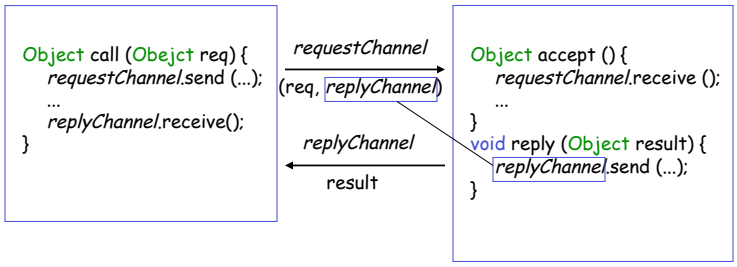
UTA

Rendezvous in Java

```
class Entry {
    private Port requestChannel = new Port();
    private CallMsg cm;
    public Object call(Object request) throws InterruptedException {
        Link replyChannel = new Link();
        requestChannel.send(new CallMsg(request, replyChannel));
        return replyChannel.receive();
    }
    public Object call() throws InterruptedException {
        Link replyChannel = new Link();
        requestChannel.send(new CallMsg(replyChannel));
        return replyChannel.receive();
    }
    public Object accept() throws InterruptedException {
        cm = (callMsg) requestChannel.receive();
        return cm.request;
    }
    public void reply(Object response) throws InterruptedException {
        cm.replyChannel.send(response);
    }
    public void reply() throws InterruptedException {
        cm.replyChannel.send();
    }
    public Object acceptAndReply() throws InterruptedException {
        cm = (CallMsg) requestChannel.receive();
        cm.replyChannel.send(new Object());
        return cm.request;
    }
    private class CallMsg {
        Object  request;
        Link replyChannel;
        CallMsg(Object m, Link c) { request=m; replyChannel=c; }
        CallMsg(Link c) { request = new Object(); replyChannel=c;}
    }

}
```

Rendezvous in Java (cont'd)



Why selective wait?

Implementation 1

```
while (true) {
   if (buffer is not full) {
      item = deposit.acceptAndReply ();
      ...
   }
   if (buffer is not empty) {
      withdraw.accept ();
      ...
      withdraw.reply (item);
   }
}
```

Implementation 2

```
while (true) {
   if (buffer is not empty) {
      withdraw.accept ();
      ...
      withdraw.reply (item);
   }
   if (buffer is not full) {
      item = deposit.acceptAndReply ();
      ...
   }
}
```

Any problem with these two implementations?

UTA

selective wait

A selective wait allows a thread to wait for a group of entries. Each entry is associated with a guard condition, which determines if the entry is acceptable.

If one or more of the entries become acceptable, then one of them will be selected for execution.

```
select
   when (the buffer is not full) => accept a call to entry deposit
or
   when (the buffer is not empty) => accept a call to entry withdraw
```


UTA

Delay and else alternatives

A selective wait can optionally contain either a delay or wait alternative:

- A delay alternative is selected if no entry can be accepted within a specified amount of time
- An else alternative is executed if no entry is acceptable.

UTA

A Java SelectiveWait (1)

1. Create a SelectiveWait object
2. Create one or more entries and then add them to SelectiveWait object
3. Create and add a delay or else alternative if necessary
4. Associate each entry with a guard condition
5. Use choose() to select one of the entries

UTA

A Java SelectiveWait (2)

```
SelectiveWait select = new SelectiveWait ();
SelectableEntry deposit = new SelectableEntry ()
SelectableEntry withdraw = new SelectableEntry ();
select.add(deposit);
select.add(withdraw);
SelectiveWait.DelayAlternative delayAlt = new SelectiveWait.DelayAlternative(500);
select.add(delayAlt);
(SelectiveWait.ElseAlternative elseAlt = new SelectiveWait.ElseAlternative();
select.add(elseAlt);)
deposit.guard (fullSlots < capacity);
withdraw.guard (fullSlots > 0);
delayAlt.guard (true);
switch (select.choose ()) {
   case 1:  deposit.acceptAndReply ();
            ...
            break;
   case 2:  withdraw.accept ();
            ...
            withdraw.reply (value);
            break;
   case 3:  delayAlt.accept ();
            break;
}
```

A Java SelectiveWait (3)

With no delay or else alternatives:

- choose will select an open alternative that has a waiting call
- if several alternative are open and have a waiting call, choose selects the one whose call arrived first.
- if one or more alternatives are open but none have a waiting call, then choose blocks

A Java SelectiveWait (4)

With else or delay alternative:

- an else alternative is executed if all the alternatives are closed or no open alternatives have a waiting call
- an open delay alternative is selected when no open alternative can be selected prior to the expiration time

UTA

A Java SelectiveWait (4)

```
final class boundedBuffer extends Thread {
        private selectableEntry deposit, withdraw;
        private int fullSlots=0;
        private int capacity = 0;
        private Object[] buffer = null;
        private int in = 0, out = 0;
        public boundedBuffer(selectableEntry deposit, selectableEntry withdraw, int capacity) {
                this.deposit = deposit; this.withdraw = withdraw;
                this.capacity = capacity;
                buffer = new Object[capacity];
        }
        public void run() {
                try {
                        selectiveWait select = new selectiveWait();
                        selectiveWait.delayAlternative delayA = select.new delayAlternative(500);
                        select.add(deposit);                                    // alternative 1
                        select.add(withdraw);           // alternative 2
                        select.add(delayA);             // alternative 3
                        while(true) {
                                withdraw.guard(fullSlots>0);
                                deposit.guard (fullSlots<capacity);
                                delayA.guard(true);
                                switch (select.choose()) {
                                        case 1:         Object o = deposit.acceptAndReply();
                                                        buffer[in] = o;
                                                        in = (in + 1) % capacity;
                                                        ++fullSlots;
                                                        break;
                                        case 2:         withdraw.accept();
                                                        Object value = buffer[out];
                                                        withdraw.reply(value);
                                                        out = (out + 1) % capacity;
                                                        --fullSlots;
                                                        break;
                                        case 3:     delayA.accept();
                                                    System.out.println("delay selected");
                                                        break;
                                }
                        }
                } catch (InterruptedException e) {}
                  catch (SelectException e) {System.out.println("deadlock detected"); System.exit(1); }
        }
}
```


UTA

Message Passing

- Introduction
- Comm. Channels
- Rendzvous and Selective Waits
- Logical Timestamps
- Message-Based Solutions

Event order

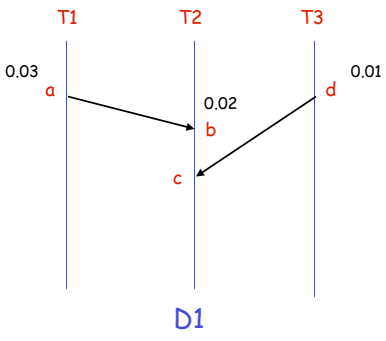
The ability to determine the order of events is important. (Example?)

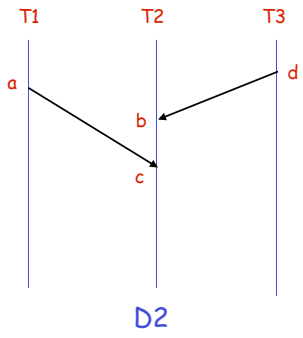
If a program is executed on a single processor, events can be timestamped using the real-time clock.

However, in a distributed environment, timestamp assignment requires clock synchronization that is difficult and sometimes impossible.

Time-space Diagram

T1	T2	T3
a. send m1 to T2	b. receive X c. receive Y	d. send m2 to T2





Incorrect & Arbitrary ordering

Assume that events are timestamped using real-time clock in D1.

- One may observe that b occurs before a if T2's clock runs a little slower than T1's clock. This is incorrect!
- Now assume that the clocks are perfectly synchronized. One may observe that d occurs before b. However, this order is arbitrary!

Logical Clock

In many cases, the use of logical clock is a better way to timestamp events.

The passage of logical time is governed by the execution of program operations, rather than the ticking of a real-time clock.

Essentially, it is the semantics of program operations that determines whether one event occurred before another event.

Happen-before Relation

The happen-before relation (⇒) preserves the causality relation between events.

- If events e and f are events in the same thread and e occurs before f, then e ⇒ f.
- If e → f, then e ⇒ f.
- If e ↔ f or f ↔ e, then for event g such that e ⇒ g, we have f ⇒ g, and for event h such that h ⇒ f, then h ⇒ e.
- if e ⇒ f and f ⇒ g, then e ⇒ g.

Happen-before Relation (cont'd)

The happen-before relation essentially defines a partial order, as it is possible that for two events e and f, neither e ⇒ f nor f ⇒ e. In this case, e and f are said to be concurrent, denoted as e ‖ f.

Question: Is ‖ transitive?

Integer Timestamps

Let P be an asynchronous message-passing program consisting of threads T1, T2, ..., and Tn. Each thread Ti, $1 \le i \le n$, contains a logical clock Ci, which is simply an integer variable initialized to 0.

- Ti increments Ci by one immediately before each event of Ti.
- When Ti sends a message, it also sends the value of Ci as the timestamp for the send event.
- When Ti receives a message with ts as its timestamp, Ti sets Ci to max(Ci, ts + 1).

Integer Timestamps (cont'd)

Let e and f be two events of an execution. Let IT(e) and IT(f) be e' and f's timestamps respectively. If $e \Rightarrow f$, then $IT(e) < IT(f)$.

Is it true that if $IT(e) < IT(f)$, then $e \Rightarrow f$?

UTA

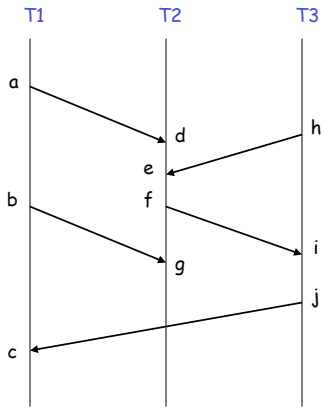
Integer Timestamps (cont'd)

Integer timestamps can be used to produce one or more total orders that preserve the causal order:

- Order the events in ascending order of their integer timestamps;
- For the events with the same integer timestamp, break the tie in some consistent way.

UTA

Integer Timestamps (cont'd)



Vector Timestamps

Integer timestamps solve the problem of incorrect ordering. However, it does not solve the problem of arbitrary ordering.

In another word, integer timestamps cannot be used to determine that two events are not causally related.

Vector Timestamps (cont'd)

Let P be a message-passing program consisting of threads T1, T2, …, and Tn.

Each thread contains a vector clock VCi, which is a vector of integer clock values. Let VCi[j], $1 \le i,j \le n$, be the jth element of VCi.

VCi[i] is similar to the logical clock Ci used for computing integer timestamp. VCi[j], where $j \neq i$, denotes the best estimate Ti is able to make about Tj's current logical clock, i.e., VCj[j].

UTA

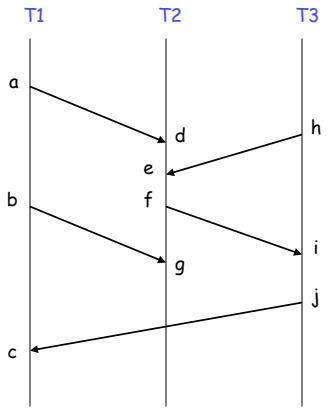
Vector Timestamps (cont'd)

VCi, $1 \le i,j \le n$, is initialized to a vector of zeros, and is maintained as follows:

- Ti increments VCi[i] by one before each event of Ti.
- When Ti executes a non-blocking send, it also sends the value of VCi as the timestamp for the send event.
- When Ti receives a message with timestamp VTm from a non-blocking send, Ti sets VCi to max(VCi, VTm), and assigns VCi as the timestamp for the receive event. Note that for (k = 1; k ≤ n; k ++) VCi[k] = max(VCi[k], VTm[k]).
- When Ti executes a blocking send to Tj or receives a message from a blocking send of Tj, Ti and Tj exchange their vector clock values, set their vector clocks to the maximum of the two vector clock values, and assign the new vector clock as the timestamp for the send and receive events.

UTA

Vector Timestamps (cont'd)



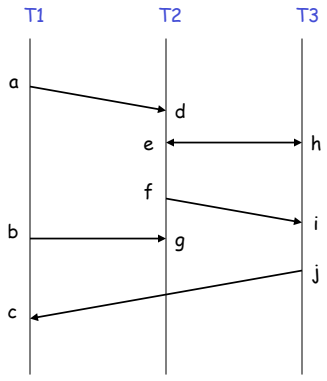
Vector Timestamps (cont'd)

In general, Ei ⇒ Ej, iff for $1 \le k \le n$, VT(Ei)[k] ≤ VT(Ej)[k] and VT(Ei) ≠ VT(Ej).

If $i \neq j$, Ei ⇒ Ej, iff VT(Ei)[i] ≤ VT(Ej)[i] and VT(Ei)[j] < VT(Ej)[j].

If only asynchronous communication is used, then Ei ⇒ Ej, iff VT(Ei)[i] ≤ VT(Ej)[i].

Vector Timestamps (cont'd)



UTA

Message Passing

- Introduction
- Comm. Channels
- Rendzvous and Selective Waits
- Logical Timestamps
- Message-Passing Based Solutions

UTA

Distributed ME

In many cases, two or more distributed processes need mutually exclusive access to some resource.

This is referred to as the distributed ME or CS problem. It needs to satisfy the same three requirements. (What?)

UTA

Permission-based Algorithm

A permission-based algorithm to the distributed ME problem is as follows:

- When a process wants to enter its CS, it sends a request to each of the other processes, and waits for each of them to reply.
- When a process receives a request, it does the following: (a) if the process is not interested in entering its CS, it gives its permission immediately; (b) otherwise, it grants its permission according to some priority scheme.

UTA

Priority Scheme

One priority scheme is to associate each request with an integer timestamp, or a sequence number.

Assume that process i receives a request from process j. Let s be the sequence number maintained by i, and t be the sequence number in the request from j.

Priority Scheme (cont'd)

- If i is not trying to enter CS, then i replies immediately.
- Otherwise, if (t < s) or ((t == s) and j < i), then i replies immediately.
- Otherwise, i defers the reply.

Distributed R==W.2

When a process wants to perform its read or write operation, it sends a request to each of the other processes and waits for each of them to reply.

Each request consists of a tuple (sequence number, thread ID, flag), where the flag indicates the type of operation being requested.

UTA

Distributed R==W.2 (cont'd)

When process i receives a request from process j, i sends j an immediate reply if:

- i is not executing or requesting to execute any operation; or
- i is executing or requesting to execute a "compatible" operation; or
- i is requesting to execute a non-compatible operation but j's request has a priority over i's request.