Outline

Time Management
- Introduction
- Message Ordering
- HLA Time Management Services
- Synchronising Wallclock time

Object/Ownership Management
- Push
- Pull
- Ownership Management in Air Traffic Control
Why Time Management?

- Delay in messages can result in wrong interpretation

- C sees target destroyed before being fired
- Temporal anomaly – can be eliminated by delaying delivery of “destroyed” event to C
The HLA provides two types of message ordering:

- **receive order (unordered):** messages passed to federate in an arbitrary order
- **time stamp order (TSO):** sender assigns a time stamp to message; successive messages passed to each federate have non-decreasing time stamps

<table>
<thead>
<tr>
<th>Property</th>
<th>Receive Order (RO)</th>
<th>Time Stamp Order (TSO)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Latency</td>
<td>low</td>
<td>higher</td>
</tr>
<tr>
<td>reproduce before and after relationships?</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>all federates see same ordering of events?</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>execution repeatable?</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>typical applications</td>
<td>training, T&amp;E</td>
<td>analysis</td>
</tr>
</tbody>
</table>

- receive order minimizes latency, does not prevent temporal anomalies
- TSO prevents temporal anomalies, but has somewhat higher latency
Causal Order

- Based on Lamport’s “happens before” relationship (1978)
- Execution of simulator viewed as ordered sequence of actions
  - E.g. sending/receiving a message

Rules:
- If actions A and B occur in the same simulator, and A appears before B in the ordered sequence of actions within that simulator, then A -> B (i.e. A happens before B)
- If A is a message send, and B is receiving the same message, then A -> B
- If A -> B, and B -> C, then A -> C (transitivity)
Causal Order

- Two events which do not have the ‘happens before’ relationship are said to be concurrent

“Happens before” (E1 -> E4, but E1 and E3 are concurrent)
Causal Order

- Two events which do not have the ‘happens before’ relationship are said to be concurrent.
- A causal order delivery service guarantees that if E1 -> E2, then each simulator receiving both messages will have E1 delivered to it before E2 is delivered.
- Concurrent events may be delivered to the simulator in any order.
- ‘happens before’ can be extended to messages:
  - If a message M1 is sent by a simulator before the same simulator sends another message M2, then M1 -> M2.
  - If M1 is delivered to a simulator before that simulator sends another message M2, then M1 -> M2.
  - If M1 -> M2, and M2 -> M3, then M1 -> M3.
- So for tank example, because tank-fires event happens before the target-destroyed event, the distributed simulation executive will delay delivery of target destroyed event until after it has delivered the tank fired event.
Causal Order

- How to implement causal ordering? Use vector clock in a multicast group
- Consider a group of \( N \) simulators
- Each simulator \( i \) maintains a vector of counters \( C_i \) where
  - \( C_i[i] \) = number of messages simulator \( i \) has sent to the group
  - \( C_i[j] \) (\( j \neq i \)) indicates the number of group messages that have been delivered to simulator \( i \) that were sent by \( j \).
- When simulator \( i \) sends a message \( M \), it increments \( C_i[i] \) by 1 and places a vector clock time stamp on the message with its new clock value
- The vector clock time stamp indicates what messages had been delivered to the sender and what other messages the sender has sent, prior to sending \( M \).
- The time stamp therefore provides a summary of the no. of messages sent by each simulator that causally precedes (\( \rightarrow \)) \( M \).
Causal Order

- e.g. A vector clock time stamp (5, 9, 6) on a message M sent by simulator 1 means that prior to sending M:
  - Simulator 1 had sent 4 messages to the group
  - 9 messages from simulator 2 were delivered to 1
  - 6 messages from simulator 3 were delivered to 1
- So 19 messages causally precede (→) M, so for any simulator belonging to the group, all 19 messages must be delivered before M is delivered in order to maintain the causal ordering property
In general, a message from $s$ with vector time stamp $T$ can only be delivered to $r$ if:

1. $T[s] = C_r[s] + 1$, and
2. $T[j] \leq C_r[j]$ for all $j \neq s$
Causal and Totally Ordered

- Causal order does not guarantee any order concerning concurrent events
- Two different simulators can see the same set of events in different order, leading to other temporal anomalies

Both blue planes attack red 2
Temporal anomaly from two simulators seeing the same set of events in a different order
• Problem can be solved if the causal order delivery service were strengthened so that not only are events delivered in causal order but all simulators are also guaranteed to receive messages for a common set of events in the same order

• Total order of events must be defined

• A causal and totally ordered communication service (CATOCS) ensures that
  – If E1 -> E2 and a simulator receives messages for both E1 and E2, then E1 will be delivered before E2
  – If two simulators both receive messages for any two events E1 and E2, both will receive them in the same order
Advancing Logical Time

HLA TM services define a protocol for federates to advance logical time; logical time only advances when that federate explicitly requests an advance:
- Time Advance Request: *time stepped federates*
- Next Event Request: *event stepped federates*
- Time Advance Grant: RTI invokes to acknowledge logical time advances

If the logical time of a federate is $T$, the RTI guarantees no more TSO messages will be passed to the federate with time stamp $< T$

Federates responsible for pacing logical time advances with wallclock time in real-time executions
HLA Time Management Services

- federate
  - local time and event management
  - mechanism to pace execution with wallclock time (if necessary)
  - federate specific techniques (e.g., compensation for message latencies)

Runtime Infrastructure (RTI)

- event ordering
- time synchronized delivery
- receive order messages
- time stamp order messages
- FIFO queue
- time stamp ordered queue
- state updates and interactions
- logical time advances

Logical time

Federate

Wallclock time
(synchronized with other processors)
Synchronizing Message Delivery

Goal: process all events (local and incoming messages) in time stamp order; To support this, RTI will

- Deliver messages in time stamp order (TSO)
- Synchronize delivery with simulation time advances

Federate: next local event has time stamp T

- If no TSO messages w/ time stamp < T, advance to T, process local event
- If there is a TSO message w/ time stamp T’ ≤ T, advance to T’ and process TSO message
Next Event Request (NER)

- Federate invokes **Next Event Request (T)** to request its logical time be advanced to time stamp of next TSO message, or T, which ever is smaller
- If next TSO message has time stamp \( T' \leq T \)
  - RTI delivers next TSO message, and all others with time stamp \( T' \)
  - RTI issues **Time Advance Grant (T')**
- Else
  - RTI advances federate’s time to \( T \), invokes **Time Advance Grant (T)**

### Typical execution sequences

<table>
<thead>
<tr>
<th>Federate</th>
<th>RTI</th>
<th>Federate</th>
<th>RTI</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>NER(T)</td>
<td></td>
<td>NER(T)</td>
</tr>
<tr>
<td></td>
<td>RAV (T')</td>
<td></td>
<td>TAG(T)</td>
</tr>
<tr>
<td></td>
<td>RAV (T')</td>
<td>Wall clock time</td>
<td></td>
</tr>
<tr>
<td>RTI delivers events</td>
<td></td>
<td>no TSO events</td>
<td></td>
</tr>
</tbody>
</table>

**NER:** Next Event Request  
**TAG:** Time Advance Grant  
**RAV:** Reflect Attribute Values  
Federate calls in black  
RTI callbacks in red
**Code Example: Event Stepped Federate**

<table>
<thead>
<tr>
<th>sequential simulator</th>
<th>federated simulator</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T = ) current simulation time</td>
<td>( T = ) time of next event in PES</td>
</tr>
<tr>
<td>PES = pending event set</td>
<td>PendingNER = TRUE;</td>
</tr>
<tr>
<td>While (simulation not complete)</td>
<td>NextEventRequest(T)</td>
</tr>
<tr>
<td>( T = ) time of next event in PES</td>
<td>while (PendingNER) Tick();</td>
</tr>
<tr>
<td>process next event in PES</td>
<td>process next event in PES</td>
</tr>
<tr>
<td>End-While</td>
<td>End-While</td>
</tr>
</tbody>
</table>

/* the following federate-ambassador procedures are called by the RTI */

Procedure ReflectAttributeValue (...)
place event in PES

Procedure TimeAdvanceGrant (...)
PendingNER = False;
Lookahead in the HLA

- Each federate must declare a non-negative lookahead value
- Any TSO sent by a federate must have time stamp at least the federate’s current time plus its lookahead
- Lookahead can change during the execution (*Modify Lookahead*)
  - increases take effect immediately
  - decreases do not take effect until the federate advances its logical time

1. Current time is T, lookahead L
2. Request lookahead decrease by $\Delta L$ to $L'$
3. Advance $\Delta T$, lookahead, decreases $\Delta T$
4. After advancing $\Delta L$, lookahead is $L'$
Federate/RTI Guarantees

Federate at logical time T (with lookahead L)
- All outgoing TSO messages must have time stamp ≥ T+L (L>0)

Time Advance Request (T)
- Once invoked, federate cannot send messages with time stamp less than T plus lookahead

Next Event Request (T)
- Once invoked, federate cannot send messages with time stamp less than T plus the federate’s lookahead unless a grant is issued to a time less than T

Time Advance Grant (T) (after TAR or NER service)
- All TSO messages with time stamp less than or equal to T have been delivered
Event Retraction

Previously sent events can be “unsent” via the Retract service
- Update Attribute Values and Send Interaction return a “handle” to the scheduled event
- Handle can be used to Retract (unschedule) the event
- Can only retract event if its time stamp > current time + lookahead
- Retracted event never delivered to destination (unless Flush Queue used)

Sample execution sequence:

1. Vehicle schedules position update at time 100, ready to advance to time 100
2. receives interaction (break down event) invalidating position update at time 100
3. Vehicle retracts update scheduled for time 100
Optimistic Time Management in the HLA

<table>
<thead>
<tr>
<th>HLA Support for Optimistic Federates</th>
</tr>
</thead>
<tbody>
<tr>
<td>• federations may include conservative and/or optimistic federates</td>
</tr>
<tr>
<td>• federates not aware of local time management mechanism of other federates (optimistic or conservative)</td>
</tr>
<tr>
<td>• optimistic events (events that may be later cancelled) will not be delivered to conservative federates that cannot roll back</td>
</tr>
<tr>
<td>• optimistic events can be delivered to other optimistic federates</td>
</tr>
<tr>
<td>• individual federates may be sequential or parallel simulations</td>
</tr>
</tbody>
</table>

**Flush Queue Request:** similar to NER except
(1) deliver all messages in RTI’s local message queues,
(2) need not wait for other federates before issuing a Time Advance Grant
Summary: HLA Time Management

Functionality:
• allows federates with different time management requirements (and local TM mechanisms) to be combined within a single federation execution
  – DIS-style training simulations
  – simulations with hard real-time constraints
  – event-driven simulations
  – time-stepped simulations
  – optimistic simulations

HLA Time Management services:
• Event order
  – receive order delivery
  – time stamp order delivery
• Logical time advance mechanisms
  – TAR
  – NER
• In a distributed simulation, each simulator clock must be synchronised
• True time – UTC (universal time coordinated)
• System may be self-contained, hardware clocks synchronised amongst themselves, no need to synchronise with UTC
• System interacts with devices synchronised with UTC, e.g. hardware-in-the-loop simulation, clocks must synchronise with UTC
Synchronisation of hardware clocks

- Assumptions
  - nodes may have different times
  - no clock can run backward
  - changes must be introduced gradually
Synchronisation of hardware clocks

- synchronization algorithms
  - Cristian’s algorithm (centralised pull method)
    - a time-server, S, has exact time (Universal Coordinated Time, UTC)
    - P requests for time at T0 and gets response at T1
    - P assumes message propagation is \((T1-T0)/2\)

* Berkeley algorithm (centralised push)
  ° coordinator collects client times and calculates mean time
  ° tells clients to speed up or slow down
Cristian's Algorithm

- Getting the current time from a time server.

Both \( T_0 \) and \( T_1 \) are measured with the same clock.

Client

Time server

Request

\( C_{UTC} \)

\( T_0 \) to \( T_1 \)

\( I \), Interrupt handling time

Time
The Berkeley Algorithm

a) The time daemon asks all the other machines for their clock values
b) The machines answer
c) The time daemon tells everyone how to adjust their clock
Object and Ownership Management in HLA

- Addresses issues concerning creation, registration, distribution, discovery, … of objects

- Also consistency of object’s data when:
  - transferred between models with different aggregation levels
  - ownership of attributes of objects is transferred from federate to another
Ownership Management in HLA

• Enables federates to share and transfer ownership of attributes of objects

• Two methods
  • Pull
  • Push(conditional, unconditional)

• Restriction
  • No single attribute of an object can be owned by two federates at the same time
Ownership Management

• In the Pull approach the federate that wishes to take over the ownership of a given attribute sends a request to the RTI and initiates the process.

Ownership Transfer (Pull)

Federate A  
attributeOwnershipReleaseResponse

RTI  
requestAttributeOwnershipRelease
attributeOwnershipAcquisitionNotification

Federate B  
attributeOwnershipAcquisition
In the Push approach the federate that owns the attribute initiates the ownership exchange by informing the RTI of its willingness to give away.

- Unconditional Push

Federate A

unconditionalAttributeOwnership
Divestiture

RTI

attributeOwnershipAcquisition
Notification

Federate B

attributeOwnershipAcquisition
Ownership Management

- Conditional Push

- In the Push approach the federate that owns the attribute initiates the ownership exchange by informing the RTI of its willingness to give away.
Air Traffic Control Federation

- Homogeneous simulation
- Federates are airports

- Simulation objects (entities) are aircraft
- Each airport controls a set of aircraft within its radar range (airspace)
- Airports are responsible for updating and publishing info about aircraft they control
- Aircraft get a destination, take off, fly to that destination, land, choose a new destination, take off and fly to the new destination,…
- Aircraft must be controlled or monitored at all time during their flight.
Ownership Transfer in ATC

- When an aircraft leaves the airspace of one airport and enters the airspace of another one, the ownership of the aircraft must be transferred.
- No aircraft is left unmonitored at any time (as it is in the real life)
- The airspace of different airports overlap (as it does in the real life)
Ownership Transfer in ATC

**Pull**

- When an airport discovers that an aircraft has entered its airspace, it sends an ownership transfer request to RTI.
- RTI informs the owner airport
- If certain conditions are satisfied the owner accepts the transfer and gives up the ownership.
- RTI ask the requesting federate to start updating and publishing the attributes
Since the airspace of two adjacent airports overlap a former may try to take over the ownership again (oscillation effect)
Simulation Results
Fujitsu AP3000 with 32 nodes (unix)

- Number of airports = 2, Number of aircraft = 10,
- Avg speed = 200km/hr
- **Width of intersection area**
  - oscillations
    - 200 km: 6
    - 1200 km: 40
    - 2500 km: 63

Pull and undesirable oscillation effect
Solutions

• don’t give up the ownership (wants another airport to take over)
  • pending request left unanswered
  • There is no method in RTI to allow a federate to reject a request and the request does not disappear if the owner does not respond to it.

• time-out by requesting federate

• time-out by RTI

• wait a period before requesting ownership

• don’t request when aircraft moving away
## Results

<table>
<thead>
<tr>
<th>No. of airports</th>
<th>No. of aircraft/airport</th>
<th>Total no. of pending requests</th>
<th>Total no. of ownership exchanges</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>5</td>
<td>0</td>
<td>19</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>0</td>
<td>39</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>7</td>
<td>36</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td>19</td>
<td>66</td>
</tr>
</tbody>
</table>
- Use different sizes for subscription and publishing
The publication and subscription regions have different sizes.
The controller airport sends a negotiated divestiture when an aircraft leaves its subscription region.
RTI asks all the other federates whether any of them wants to take over.
The airport(s) who has got the aircraft within its subscription region will respond.
If several federates have responded RTI answers to the last received request.

+ The owner initiates the transfer.
- RTI decides the new owner. In a real scenario the current owner decides.
• FDK stands for Federated Simulations Development Kit which is a software system developed by Georgia Tech research group. (http://www.cc.gatech.edu/computing/pads/fdk)
• The FDK is carefully designed so that RTI developers can pick and choose from the set of FDK modules that are most appropriate for developing their particular RTI implementation.
• Implement OWM on FDK - three way handshaking
Two methods for transferring ownership in HLA

The handshaking procedures in RTI (Ownership Management) are not complete -
- no mechanism to reject attribute release request
- problem with pending requests

In simulations like ours the “Push” method is to be preferred
- amount of time spent by federates on checking conditions for pull is greater than that for push

Problems with push method
- more communication for large numbers of objects
- regions have to be chosen carefully
- RTI chooses the new owner
Distributed Virtual Environments: Summary

• Perhaps the most dominant application of distributed simulation technology to date
  – Human in the loop: training, interactive, multi-player video games
  – Hardware in the loop

• Managing interprocessor communication is the key
  – Dead reckoning techniques
  – Data distribution management

• Real-time execution essential

• Many other issues
  – Terrain databases, consistent, dynamic terrain
  – Real-time modeling and display of physical phenomena
  – Synchronization of hardware clocks
  – Human factors