Timed Systems

INF2140 Parallel Programming: Lecture 12

May 2, 2012

Timed Systems

- Concepts:
  - discrete time
  - ticks from a global clock
  - timing consistency: time-stop
- Models
  - relative speed of processes
  - maximal progress
  - output intervals, jitter, timeout
- Practice
  - thread-based vs. event-based models
  - timeout
  - asynchronous tasks
  - timed objects and time manager
  - two-phase clock

Modelling Timed Systems

- Processes execute at arbitrary relative speeds (Chap. 3)
- Delay between two actions can take arbitrarily long time
- How can we make processes aware of the passage of time?
- How can processes synchronize execution with time?
- Simplification: Assume that execution time of actions is negligible compared to external events.
- **Example:** What is the difference between *two single clicks* of a mouse, and a *double click*?
- Discrete model of time
- Passage of time signalled by a "tick" from a global clock
- The processes share the ticks

Doubleclick

```
DOUBLECLICK(D=3) =
  (tick ->  DOUBLECLICK
  |click -> PERIOD[1]
  ),

PERIOD[t:1..D] =
  (when (t==D) tick -> DOUBLECLICK
  |when (t<D)  tick -> PERIOD[t+1]
  |click -> doubleclick -> DOUBLECLICK
  ).
```

The model does not say anything about the intervals between ticks.
(Here, it could be every second. In a hardware model, it could be nanoseconds.)

More accuracy = more ticks between the clicks
= more states in the model

Timing Consistency

We now look at timing in models with multiple processes

- Producer: produces an item every Tp seconds
- Consumer: takes an item every Tc seconds

Are the timings of the two processes compatible?

Timed Producer Consumer

Producer: produce item, wait *Tp* ticks, repeat:

```
PRODUCER(Tp=3) =
  (item -> DELAY[1]),
DELAY[t:1..Tp] =
  (when(t==Tp) tick -> PRODUCER
  |when(t<Tp)  tick -> DELAY[t+1]
  ).
```

Consumer: let time pass, consume item, wait *Tc* ticks, repeat:

```
CONSUMER(Tc=3) =
  (item -> DELAY[1] | tick -> CONSUMER),
DELAY[t:1..Tc] =
  (when(t==Tc) tick -> CONSUMER
  |when(t<Tc)  tick -> DELAY[t+1]
  ).
```

Timed Producer Consumer

Can we express different relative speeds of Producer and Consumer?

Case 1: Producer and Consumer work at the same speed: $Tp = Tc = 2$

```
||SAME   = (PRODUCER(2) || CONSUMER(2)).
```

Case 2: Producer is slower than Consumer: $Tp = 3$, $Tc = 2$

```
||SLOWER = (PRODUCER(3) || CONSUMER(2)).
```

Case 3: Producer is faster than Consumer: $Tp = 2$, $Tc = 3$

```
||FASTER = (PRODUCER(2) || CONSUMER(3)).
```

Why do we get a deadlock here?

Time-stop: Deadlocks caused by timing inconsistencies

If the composed system does not produce time-stop, we say the timing assumptions of the processes are *consistent*

Maximal Progress

Assume now that we have a *store* with a certain capacity. Producer fills the store with items and Consumer takes items from the store. Let Producer and Consumer have the same *rate*.

```
STORE(N=3) = STORE[0],
STORE[i:0..N] = (put -> STORE[i+1]
                |when(i>0) get -> STORE[i-1]
                ).
||SYS = (PRODUCER(1)/{put/item}
        ||CONSUMER(1)/{get/item}
        ||STORE
        ).
```

We get a safety violation! Why?

Consumer can always choose to let time pass.

Maximal Progress

- If Consumer always chooses to let time pass, the store will overflow.
- We must ensure that an action occurs as soon as all participants are ready to do it.
- This is called *maximal progress*.
- In FSP, we can ensure maximal progress using action priority

```
||NEW_SYS = SYS>>{tick}.
```

- After a tick, all actions that can occur, will happen before the next tick

Maximal Progress

- We have assumed that the execution time of actions is negligible
- However, we don't want to allow infinitely many actions before time progresses
- We can check that time must eventually progress. How?

Define a progress property and check for progress:

```
progress TIME = {tick}
```

Maximal Progress

Is time guaranteed to progress here? Why/why not?

```
PROG = (start -> LOOP | tick -> PROG),
LOOP = (compute -> LOOP |tick -> LOOP).
||CHECK = PROG>>{tick}.

progress TIME = {tick}
```

Is time guaranteed to progress here? Why/why not?

```
PROG2 = (start -> LOOP | tick -> PROG2),
LOOP =  (compute -> LOOP |tick -> LOOP
        |end -> tick -> PROG2).
||CHECK2 = PROG2>>{tick}.

progress TIME = {tick}
```

This loop models a finite number of iterations.

Output in an Interval

- This simple model of global ticks is surprisingly expressive
- In a system with congestion, response time may vary
- How can we model that output occurs within atime interval?

```
OUTPUT(Min=1,Max=3) =
  (start -> OUTPUT[1] | tick -> OUTPUT),
OUTPUT[t:1..Max] =
  (when (t>Min && t<=Max) output -> OUTPUT
  |when (t<Max)           tick   -> OUTPUT[t+1]
  ).
```

- Output can occur at any time between Min and Max ticks

Jitter

- Output occurs at a predictable rate but at unpredictable times
- This timing uncertainty is called **jitter** in comm. systems
- Jitter = periodic output at any time in a time interval

```
JITTER(Max=2) =
  (start -> JITTER[1] | tick -> JITTER),
JITTER[t:1..Max] =
  (output -> FINISH[t]
  |when (t<Max)  tick   -> JITTER[t+1]
  |when (t==Max) output -> FINISH[t]),
FINISH[t:1..Max] =
  (when (t<Max)  tick -> FINISH[t+1]
  |when (t==Max) tick -> JITTER).
```

Timeout

- Timeout can be modelled in FSP by using a separate process
- Action setTO starts the timer
- Action resetTO stops the timer

```
TIMEOUT(D=1)
  = (setTO           -> TIMEOUT[0]
    |{tick,resetTO} -> TIMEOUT),
TIMEOUT[t:0..D]
  = (when (t<D) tick     -> TIMEOUT[t+1]
    |when (t==D)timeout -> TIMEOUT
    |resetTO             -> TIMEOUT).
```

Timeout

```
TIMEOUT(D=1)   = (setTO            -> TIMEOUT[0]
                 |{tick,resetTO} -> TIMEOUT),
TIMEOUT[t:0..D]
  = (when (t<D) tick     -> TIMEOUT[t+1]
    |when (t==D)timeout -> TIMEOUT
    |resetTO             -> TIMEOUT).

RECEIVE = (start -> setTO -> WAIT),
WAIT    = (timeout -> RECEIVE
          |receive -> resetTO -> RECEIVE).

||RECEIVER(D=2) = (RECEIVE || TIMEOUT(D))
               >>{receive,tick,timeout,start}
               @{receive,tick,timeout,start}.
```

- Give interface actions low priority, so system using RECEIVER has priority for other (regular) actions

Java Implementation

Two approaches to implementing discrete time in Java

- Thread-based approach :
  - The global clock is passive
  - Most similar to the previous chapters of the book
- Event-driven approach:
  - The global clock is active
  - Time advance triggers activities in the objects
  - Approach discussed in the book

Thread-based Approach

If we are working with machine-time:

- sleep(long ms)
  - can be called on a thread
  - causes the thread to suspend execution for ms milliseconds
- wait(long ms)
  - can be called on a lock
  - causes the thread to suspend execution until it is either notified on the lock or ms milliseconds have passed
  - when the thread resumes execution, it does not know if it woke up from the timeout or not

Timeout Monitor

```
class TimeoutMonitor {
  boolean notified = false;

  public synchronized boolean timer(long ms)
  throws InterruptedException {
    if (!notified) wait(ms);
    // Client reactivated with notification: notified
    return notified;
  }
  public synchronized void alert(){
    notified= true;
    notifyAll();
    // Notified monitor
  }
}
```

Combining Timeout with Computation

- Idea: Make the execution *asynchronous*
- Let a Client class spawn a Server class which extends Thread

```
public class Server
extends Thread implements Runnable {
  TimeoutMonitor tm; String result = "no result";

  public Server(TimeoutMonitor timeoutmon){
    tm=timeoutmon; }
  public String get(){ return result; }
  public void run() {
    try { Thread.sleep(4000); // Long exec. time
      result = "execution completed";
      tm.alert(); // Server completed execution
    } catch (InterruptedException e) { }
}}
```

Combining Timeout with Computation

```
public class Client {
  public static String result="";

  public  static void main(String[] args){
    try {
      TimeoutMonitor tm = new TimeoutMonitor();
      Server ts = new Server(tm);
      ts.start();                               // start job
      boolean noTimeout=tm.timer(10); // start timer
      if (noTimeout) result=ts.get(); // get result
    } catch (InterruptedException e) {}
    // Finished execution with result: result
    }
}
```

Asynchronous Tasks

- We can generalize this picture, using **asynchronous tasks**
- The Server class of our example is essentially an Executor
- The run method of Server is a Task to be executed
- The result of a Task execution, is stored in a Future object
- If the return value of the Task is a type T, the associated future will have the type Future<T>
- Method to access return value: future.get()

Asynchronous Tasks

```java
public class AsyncCall {
  public static void main(String[] args)
  throws Exception {
    ExecutorService executor =
      Executors.newSingleThreadExecutor();
    Future<String> future =
      executor.submit(new Task()); // Start execution
    String result = future.get();  // Wait for result
    System.out.println(result);
    executor.shutdownNow();
  }
}
class Task implements Callable<String> {
  public String call() throws Exception {
    Thread.sleep(4000); return "Ready!"; }
}
```

Asynchronous Tasks with Timeout

```java
public class TimeOutTest {
  public static void main(String[] args)
  throws Exception {
    ExecutorService executor =
      Executors.newSingleThreadExecutor();
    Future<String> future =
      executor.submit(new Task()); // Start execution
    try {
      String result = future.get(5, TimeUnit.SECONDS);
      System.out.println(result); // Finish correctly
   } catch (TimeoutException e) {...} // Timeout!
   }
    executor.shutdownNow();
}}
```

Event-driven Approach: Timed Objects

- Advantage compared to thread-based approach:
  avoids context-switching.
- **Basic Idea**: A TimeManager triggers activities in the objects
- Timed Objects: These activities follow a predefined order
- TimeStop: exception to catch timing inconsistencies

```
public interface Timed {
  void pretick() throws TimeStop;
  void tick();
}
```

Countdown Timer

Simple FSP model

```
COUNTDOWN (N=3)    = COUNTDOWN[N],
COUNTDOWN[i:0..N] =
  (when(i>0)  tick -> COUNTDOWN[i-1]
  |when(i==0) beep -> STOP
  ).
```

Countdown Timer

```
class TimedCountDown implements Timed {
  TimeManager clock; int i;

  TimedCountDown(int N, TimeManager clock) {
    i = N; this.clock = clock;
    clock.addTimed(this); // Get ticks
  }
  public void pretick() throws TimeStop {
    if(i==0) { ...                        // Do beep action
        clock.removeTimed(this); } // No more ticks
  }
  public void tick(){ --i; }
}
```

Timed Producer Consumer

Producer: produce item, wait *Tp* ticks, repeat:

```
PRODUCER(Tp=3) =
  (item -> DELAY[1]),
DELAY[t:1..Tp] =
  (when(t==Tp) tick -> PRODUCER
  |when(t<Tp)  tick -> DELAY[t+1]
  ).
```

Consumer: let time pass, consume item, wait *Tc* ticks, repeat:

```
CONSUMER(Tc=3) =
  (item -> DELAY[1] | tick -> CONSUMER),
DELAY[t:1..Tc] =
  (when(t==Tc) tick -> CONSUMER
  |when(t<Tc)  tick -> DELAY[t+1]
  ).
```

Timed Producer-Consumer

```
class ProducerConsumer {
  TimeManager clock = new TimeManager(1000);
  Producer producer = new Producer(2);
  Consumer consumer = new Consumer(3);

  ProducerConsumer() {clock.start();}

  class Producer implements Timed { ... }
  class Consumer implements Timed { ... }
}
```

Timed Producer

```
class Producer implements Timed {
  int Tp,t;

  Producer(int Tp) {
    this.Tp = Tp;  t=1;
    clock.addTimed(this);
  }
  public void pretick() throws TimeStop {
    if (t==1) consumer.item(new Object());
  }
  public void tick() {
    System.out.println("Tick producer");
    if (t<Tp) { ++t;return;}
    if (t==Tp) {t=1;}
  }
}
```

Timed Consumer

```
class Consumer implements Timed {
  int Tc,t; Object consuming = null;

  Consumer(int Tc) {
    this.Tc = Tc; t=1; clock.addTimed(this);
  }
  void item(Object x) throws TimeStop {
    // ...println("Transfer");
    if (consuming!=null) throw new TimeStop();
    consuming = x; }
  public void pretick() {}
  public void tick() {
    // ...println("Tick consumer "+(consuming!=null));
    if (consuming==null) return;
    if (t<Tc) { ++t; return;}
    if (t==Tc) {consuming=null; t=1;}}}
```

Time Manager

```
public class TimeManager extends Thread {
  int delay; // clocked: why volatile? why immutable?
  volatile ImmutableList<Timed> clocked = null;

  public TimeManager(int d) {delay = d;}

  public void addTimed(Timed el) {
    clocked = ImmutableList.add(clocked,el);
  }
  public void removeTimed(Timed el) {
    clocked = ImmutableList.remove(clocked,el);
  }
  public void run () { ... }
}
```

Time Manager

```
public class TimeManager extends Thread { ...
 public void run () {
    try {
      while(true) {
        try { // broadcast of pretick and tick
          for (Timed e: clocked) e.pretick();
          for (Timed e :clocked) e.tick();
        } catch (TimeStop s) {
            System.out.println("*** TimeStop");
            return;
        }
        Thread.sleep(delay);
      }
    } catch (InterruptedException e){}
}}
```

Two-Phase Clock

- In FSP: Maximal progress
- In Java: Two-phase clock (pretick & tick)
- Each Timed object has one opportunity to perform an action: the pretick
- A multi-way interaction between objects will require several clock-cycles (in contrast to FSP)
- Although it can be done in one cycle in FSP, we must take care that multiway interaction in one cycle is not required in Java (e.g., by introducing a tick between a request and the reply)
- Thread-based approach
  - looser coupling between time and execution in Java
  - to implement logical (program) time, let the TimeOutMonitor class be a Timed Object (and use wait() instead of wait(ms))

Timed Systems: Summary

- Concepts:
  - discrete time
  - ticks from a global clock
  - timing consistency: time-stop
- Models
  - relative speed of processes
  - maximal progress
  - output intervals, jitter, timeout
- Practice
  - thread-based vs. event-based models
  - timeout
  - asynchronous tasks
  - timed objects and time manager
  - two-phase clock