Distributed Systems Concepts

Ch. 10 and 14-17

Figure 10.1
Skew between computer clocks in a distributed system

Network

Figure 10.2
Clock synchronization using a time server

Figure 10.3
An example synchronization subnet in an NTP implementation

Note: Arrows denote synchronization control, numbers denote strata.

Figure 10.4
Messages exchanged between a pair of NTP peers

Figure 10.5
Events occurring at three processes

Physical time
**Figure 10.6**
Lamport timestamps for the events shown in Figure 10.5

**Figure 10.7**
Vector timestamps for the events shown in Figure 10.5

**Figure 10.8**
Detecting global properties

- a. Garbage collection
- b. Deadlock
- c. Termination

**Figure 10.9**
Cuts

- Consistent cut
- Inconsistent cut

**Figure 10.10**
Chandy and Lamport’s ‘snapshot’ algorithm

Marker receiving rule for process $p_i$

- On $p_i$’s receipt of a marker message over channel $c$:
  - if ($p_i$ has not yet recorded its state) then:
    - records its process state now;
    - records the state of $c$ as the empty set;
    - turns on recording of messages arriving over other incoming channels $c$;
  - else:
    - $p_i$ records the state of $c$ as the set of messages it has received over $c$ since it saved its state.
- end if

Marker sending rule for process $p_i$

- After $p_i$ has recorded its state, for each outgoing channel $c$:
  - $p_i$ sends one marker message over $c$ (before it sends any other message over $c$).

**Figure 10.11**
Two processes and their initial states

- $p_1$: $\$1000 account, (none) widgets
- $p_2$: $\$50 account, 2000 widgets
The execution of the processes in Figure 10.11:

1. Global state $S_0$:
   - $p_1$, $p_2$ (empty)
   - $c_1$, $c_2$

2. Global state $S_1$:
   - $p_1$ (Order 10, $100$), $M$ < $900$, 0
   - $c_1$, $c_2$ (five widgets)

3. Global state $S_2$:
   - $p_1$, $p_2$ (Order 10, $100$), $M$ < $900$, 5
   - $c_1$, $c_2$ (empty)

4. Global state $S_3$:
   - $p_1$, $p_2$ (empty)
   - $c_1$, $c_2$

(M = marker message)

Reachability between states in the snapshot algorithm:

- Actual execution: $e_0, e_1, ...$
- Recording begins: $S_{init}$
- Recording ends: $S_{final}$
- Pre-snap: $e_0', e_1', ..., e_{R-1}'$
- Post-snap: $e_R', e_{R+1}', ...$

Vector timestamps and variable values for the execution of Figure 10.9:

- Physical time: Cut $C_1$ (1,0) (2,0) (4,3)
  - $x_1 = 1$
  - $x_2 = 100$
- Cut $C_2$ (2,1) (2,2) (2,3)
  - $x_1 = 90$
  - $x_2 = 95$

- Physical time: Cut $C_3$ (2,1) (2,2) (2,3)
  - $x_1 = 100$
  - $x_2 = 90$

Chapter 11, 12, and 13

- Are on transaction and concurrency control that are typically covered in a database course.

Fault Tolerance

A basic architectural model for the management of replicated data:

- Front ends
- Service
- Replica managers
- Requests and replies
- Clients
Figure 14.2 Services provided for process groups

Group address expansion
Group send
Multicast communication
Leave
Join
Process group

Group membership management

Figure 14.3 View-synchronous group communication

Figure 14.4 The passive (primary-backup) model for fault tolerance

Figure 14.5 Active replication

Ch. 15: Distributed Multimedia Systems

Figure 15.1 A distributed multimedia system
The window of scarcity for computing and communication

Figure 15.2
The window of scarcity for computing and communication

remote login
network file access
high-quality audio
interactive video

1980
1990
2000

characteristics of typical multimedia streams

<table>
<thead>
<tr>
<th>Data rate (approximate)</th>
<th>Sample or frame frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Telephone speech</td>
<td>64 kbps</td>
</tr>
<tr>
<td>CD-quality sound</td>
<td>1.4 Mbps</td>
</tr>
<tr>
<td>Standard TV video (uncompressed)</td>
<td>120 Mbps</td>
</tr>
<tr>
<td>Standard TV video (MPEG-1 compressed)</td>
<td>1.5 Mbps</td>
</tr>
<tr>
<td>HDTV video (uncompressed)</td>
<td>1000-3000 Mbps up to 1920x1080 pixels</td>
</tr>
<tr>
<td>HDTV video (MPEG-2 compressed)</td>
<td>10-30 Mbps</td>
</tr>
</tbody>
</table>

Figure 15.3
Characteristics of typical multimedia streams

Figure 15.4
Typical infrastructure components for multimedia applications

Microphones
Camera
Screen
Window system
Codec
DB
Mixer
PC/workstation
CM
Video store
Network connections

K
L
M

multimedia stream
white boxes represent media processing components, many of which are implemented in software, including codec, sound mixing component

Figure 15.5
QoS specifications for components of the application shown in Figure 15.4

<table>
<thead>
<tr>
<th>Component</th>
<th>Bandwidth</th>
<th>Latency</th>
<th>Loss rate</th>
<th>Resources required</th>
</tr>
</thead>
<tbody>
<tr>
<td>Camera Out: 10 frames/sec, raw video 640x480x16 bits</td>
<td>Zero</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A Codec In: Out: 10 frames/sec, raw video MPEG-1 stream</td>
<td>Interactive Low 10 ms CPU each 100 ms; 10 Mbytes RAM</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B Mixer In: Out: 1 44 kbps audio 1 44 kbps audio</td>
<td>Interactive Very low 1 ms CPU each 100 ms; 1 Mbytes RAM</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>H Window system In: Out: various 50 frame/sec framebuffer</td>
<td>Interactive Low 5 ms CPU each 100 ms; 5 Mbytes RAM</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>K Network connection In/Out: MPEG-1 stream, approx. 1.5 Mbps, low-loss stream protocol</td>
<td>Interactive Very low</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L Network connection In/Out: Audio 44 kbps</td>
<td>Very low 44 kbps, very low-loss stream protocol</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 15.6
The QoS manager’s task

Application components specify their QoS requirements to QoS manager

Yes
No

Resource contract

Admission control

QoS negotiation

Application components request QoS requirements to QoS manager

Traffic shaping algorithms

(a) Leaky bucket
(b) Token bucket

Figure 15.7
Traffic shaping algorithms

Token generator
Figure 15.8
The RFC 1363 Flow Spec

<table>
<thead>
<tr>
<th>Bandwidth:</th>
<th>Protocol version</th>
<th>Maximum transmission unit</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Maximum transmission rate</td>
<td>Token bucket size</td>
</tr>
<tr>
<td></td>
<td>Token bucket rate</td>
<td>Maximum transmission rate</td>
</tr>
<tr>
<td>Delay:</td>
<td>Minimum delay noticed</td>
<td>Maximum delay variation</td>
</tr>
<tr>
<td></td>
<td>Loss sensitivity</td>
<td>Burst loss sensitivity</td>
</tr>
<tr>
<td></td>
<td>Loss interval</td>
<td>Quality of guarantee</td>
</tr>
</tbody>
</table>

Figure 15.9
Filtering

Source

Targets

Highbandwidth

Mediumbandwidth

Lowbandwidth

Figure 15.10
Tiger video file server hardware configuration

Figure 15.11
Tiger schedule

Figure 16.1
The distributed shared memory abstraction

16: Distributed Shared Memory
Program Writer:
main()
{
  struct shared *p;
  methersetup(); /* Initialize the Mether run-time */
  p = (struct shared *)METHERBASE;
  /* overlay structure on METHER segment */
  p->a = p->b = 0; /* initialize fields to zero */
  while(TRUE) { /* continuously update structure fields */
    p->a = p->a + 1;
    p->b = p->b - 1;
  }
}

Continued on next slide...

Program Reader:
main()
{
  struct shared *p;
  methersetup();
  p = (struct shared *)METHERBASE;
  while(TRUE) { /* read the fields once every second */
    printf("a = %d, b = %d
", p->a, p->b);
    sleep(1);
  }
}

IDL interfaces Shape and ShapeList

struct Rectangle
{ 1
  long width;
  long height;
  long x;
  long y;
}

struct GraphicalObject
{ 2
  string type;
  Rectangle enclosing;
  boolean isFilled;
}

interface Shape
{ 3
  long getVersion();
  GraphicalObject getAllState(); // returns state of the GraphicalObject
};

typedef sequence <Shape, 100> All; 4

interface ShapeList
{ 5
  exception FullException{ }; 6
  Shape newShape(in GraphicalObject g) raises (FullException); 7
  All allShapes(); // returns sequence of remote object references 8
  long getVersion(); 9
};

class ShapeList extends org.omg.CORBA.Object { 10
  Shape newShape(GraphicalObject g) throws ShapeListPackage.FullException;
  All allShapes();
  int getVersion();
};
Figure 17.3
ShapeListServant class of the Java server program for CORBA interface ShapeList

```java
import org.omg.CORBA.*;
class ShapeListServant extends _ShapeListImplBase {
    ORB theOrb;
    private Shape theList[];
    private int version;
    private static int n=0;
    public ShapeListServant(ORB orb){
        theOrb = orb;
        // initialize the other instance variables
    }
    public Shape newShape(GraphicalObject g) throws ShapeListPackage.FullException { 1
        version++;
        Shape s = new ShapeServant( g, version);
        if(n >=100) throw new ShapeListPackage.FullException();
        theList[n++] = s; 2
        theOrb.connect(s);
        return s;
    }
    public Shape[] allShapes(){ ... }
    public int getVersion() { ... }
}
```

Figure 17.4
Java class ShapeListServer

```java
import org.omg.CORBA.*;
public class ShapeListServer {
    public static void main(String args[]) {
        ORB orb = ORB.init(args, null); 1
        ShapeListServant shapeRef = new ShapeListServant(orb); 2
        orb.connect(shapeRef); 3
        org.omg.CORBA.Object objRef =
            orb.resolve_initial_references("NameService");  4
        NamingContext ncRef = NamingContextHelper.narrow(objRef);
        NameComponent nc = new NameComponent("ShapeList", "); 5
        NameComponent path[] = { nc }; 6
        ncRef.rebind(path, shapeRef);                    7
        java.lang.Object sync = new java.lang.Object();
        synchronized (sync) {   sync.wait();}
    }
}
```

Figure 17.5
Java client program for CORBA interfaces Shape and ShapeList

```java
import org.omg.CORBA.*;
import org.omg.CORBA.*;
public class ShapeListClient{
    public static void main(String args[]) {
        try{
            ORB orb = ORB.init(args, null);
            org.omg.CORBA.Object objRef =
                orb.resolve_initial_references("NameService");
            NamingContext ncRef = NamingContextHelper.narrow(objRef);
            NameComponent nc = new NameComponent("ShapeList", ");
            NameComponent path[] = { nc }; 1
            ShapeList shapeListRef = ShapeListHelper.narrow(ncRef.resolve(path)); 2
            Shape[] sList = shapeListRef.allShapes(); 3
            GraphicalObject g = sList[0].getAllState(); 4
        } catch (Exception e) { ... }
    }
}
```

Figure 17.6
The main components of the CORBA architecture

```
client
Implementation repository
interface repository
Object adapter
or dynamic invocation
Skeleton
ORB
or dynamic skeleton
```

Figure 17.7
IDL module Whiteboard

```idl
module Whiteboard {
    struct Rectangle{
        ...
    };
    struct GraphicalObject {
        ...
    };
    interface Shape {
        ...
    };
    typedef sequence <Shape, 100> All;
    interface ShapeList {
        ...
    };
}
```

Figure 17.8
IDL constructed types – 1

<table>
<thead>
<tr>
<th>Type</th>
<th>Examples</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequence</td>
<td>typedef sequence &lt;Shape, 100&gt; All</td>
<td>Defines a type for a variable-length sequence of elements of a specified IDL type. An upper bound on the length may be specified.</td>
</tr>
<tr>
<td>string</td>
<td>typedef string &lt;8&gt; SmallString</td>
<td>Defines a type for a string of characters, terminated by the null character. An upper bound on the length may be specified.</td>
</tr>
<tr>
<td>array</td>
<td>typedef octet uniqueId[12]; typedef GraphicalObject GO[10][10]</td>
<td>Defines a type for a multi-dimensional fixed-length sequence of elements of a specified IDL type.</td>
</tr>
</tbody>
</table>

this figure continues on the next slide
**IDL constructed types – 2**

<table>
<thead>
<tr>
<th>Type</th>
<th>Examples</th>
<th>Use</th>
</tr>
</thead>
</table>
| record | struct GraphicalObject {
|        | string type;
|        | Rectangle enclosing;
|        | boolean isFilled;
|        | }                                               | Defines a type for a record containing a group of related entities. Structures are passed by value in arguments and results. |
|        |                                               |                                                                     |
| enumerated | enum Rand {
|        | Exp, Number, Name;
|        | }                                               | The enumerated type in IDL maps a type name onto a small set of integer values. |
|        |                                               |                                                                     |
| union  | union Exp switch (Rand) {
|        | case Exp: string vote;
|        | case Number: long n;
|        | case Name: string s;
|        | }                                               | The IDL discriminated union allows one of a given set of types to be passed as an argument. The header is parameterized by an enum which specifies which member is in use. |

---

**Figure 17.9**

**Naming graph in CORBA Naming Service**

**Figure 17.10**

**Part of the CORBA Naming Service NamingContext interface in IDL**

```idl
struct NameComponent {
  string id;
  string kind;
} typedef sequence <NameComponent> Name;

interface NamingContext {
  void bind (in Name n, in Object obj);
  binds the given name and remote object reference in my context.
  void unbind (in Name n);
  removes an existing binding with the given name.
  void bind_new_context (in Name n);
  creates a new naming context and binds it to a given name in my context.
  Object resolve (in Name n);
  looks up the name in my context and returns its remote object reference.
  void list (in unsigned long how_many, out BindingList bl, out BindingIterator bi);
  returns the names in the bindings in my context.
};
```

---

**Figure 17.11**

**CORBA event channels**

---

**Page 684**

**CORBA interoperable object references**

**IOR format**

<table>
<thead>
<tr>
<th>IDL interface type name</th>
<th>Protocol and address details</th>
<th>Object key</th>
</tr>
</thead>
<tbody>
<tr>
<td>interface repository</td>
<td>Protocol and address details</td>
<td>Object name</td>
</tr>
<tr>
<td>host domain</td>
<td>port number</td>
<td>adapter name</td>
</tr>
</tbody>
</table>