CptS 464/564
Lecture 25

6th December 2004

Overview

Homework #2 is posted in the assignments directory. We will discuss it next time if we don't get to it today.

Project issues

Security

Project issues

Be careful in implementing the logical clocks. Messages should be received *after* they are sent and the logical clock values should reflect this. The vector clocks discussion in the book is not very good – you may want to look up the original source material.

Implementation issues for the CO object

The CO object is a push consumer as far as the shared event channel is concerned, so it will have a push consumer servant. It is a push supplier as far as the application is concerned. How best to implement this?

The approach that I would use is to have the push consumer servant code act as the push supplier, something like this:

```
push(Any a) {
   a >> (msgClock, sender, msgData)
   put (msgClock, sender, msgData) in incoming queue
   while there are deliverable messages
      remove a deliverable message from the incoming queue
      **deliver it by calling the push method of the
        application's push consumer object
      ***
}
```

The structure of this method is reminiscent of the way we had library objects making calls on other library objects in project 2. There is a potential problem lurking, though. What if a second call to the above push() method occurs while the first is being processed?

Resolving this problem will require different techniques with different ORB implementations and choices of options. With ORBacus and the default *reactive* model option for C++ or *threaded* model for Java, we don't have to worry about *arbitrary* interleaving of execution. In both cases the ORB guarantees that only one instance of the user code is active at any time. (See Ch. 18 of the Orbacus manual)

However, at point **, while the push method of the application's push consumer is being called, additional messages might be processed received and processed. It is therefore *critical* to have clearly in mind how you are going to allocate responsibility for deliverable messages to multiple running instances of push(). I've suggested one way above: remove a message from the shared incoming queue before sending it. That way, subsequent instances of push will not see the that this instance is delivering. Likewise, each iteration of the loop needs to look at the state of the queues afresh: don't assume anything about the state of shared data at the point marked ***.

The approach described here is sufficient to meet the requirements of the project: a correctly implemented CO object will push messages to the application in causal order. However, notice that we would have to know something more about the ORB implementations to know that under all circumstances the messages would be received by the application in the same order that they were pushed.

Implementation issues for the Application

The application as specified for the project cries out to be implemented as a mixed client and server. The client piece would sporadically (or aperiodically) send messages. The server would receive messages from the EventChannel by having its *push* method called. With the facilities available in the basic ORBacus reactor model this appears to be remarkably difficult. (The ORBacus reactors for X11 and Windows might make it possible to pull it off, but that's going way beyond the intent of this class.) Doing it as a threaded application, at least for C++, is also beyond the scope of this class.

What I propose, therefore, is that we cheat. Remember the purpose of the project is to implement the CO object. The application is intended merely to demonstrate the CO object. What I'm proposing is to use a second process as the source of the application-originated messages, a role that *should* be played by a second thread.

With this concession to practicality, the main part of the application's basic structure is similar to the CO object. In one process, implement the push consumer to receive messages from the CO object. Also implement a method that receives messages from application's message driver program, prints them, and sends them on their way to the CO object.

Why not use the polling methods of the ORB?

The orb supplies two methods in addition to *run()* that might be useful for the mixed client/server program. *orb->work_pending()* returns a boolean indicating whether the ORB currently has any incoming calls to process and *orb->perform_work()* processes one incoming call.

So you could implement the application as an active client that periodically called perform_work() when work_pending() was true. But, to have your application be responsive to incoming messages you would essentially have to be constantly calling work_pending() – busy waiting – and that is so inelegant, and so potentially disruptive to other users of the lab, that I cannot recommend it.

Security concepts for distributed systems

What is security? Surprisingly, in looking at half a dozen or more chapters concerning security I found not a single definition. The closest thing to a definition

I found was from Tanenbaum, attributed to Laprie "Security is strongly related to the notion of dependability. Informally, a dependable system is one that we *justifiably trust* [emphasis added] to deliver its services." So I would argue that security is a state of mind, having to do with trust: trust that systems will do what we expect of them, when we want it done; and that they won't do things we don't want them to do. Of course we might say the same thing about *correctness* of systems. Using the word *security* adds an implication that the threats to correct operation are more deliberate or malicious than accidental. Deliberate and malicious threats also perhaps cause us to look at "the system" as encompassing more than just a computer system – people and people processes may be more vulnerable to subversion than are the computer systems that they use.

The study of fault-tolerance has produced a useful vocabulary for talking about how and why systems fail to meet expectations. There are three important terms

Failure a component does not meet (or no longer meets) its specification (externally visible)

Error a system is in an incorrect state

Fault an adjudged cause for an error

Failures are visible externally: specifications tell us how a component should behave with respect to the rest of the system. Errors may only be visible by using diagnostic tools. Faults are *latent* until they cause errors and failures. An example of a fault: code that may overrun a buffer; an associated error: the buffer has been overrun; associated failures: the process crashes (bad), the process begins to execute code controlled by an unauthorized party (worse). So causality goes fault->error->failure; diagnosis typically goes failure->error->fault.

Of course, faults and failures can cascade: a fault in A causes an error in component A's state and a failure of A. Component B, relying on A, suffers a fault leading to its own failure.

There is a more security-specific vocabulary as well:

Vulnerability non-malicious (latent) fault or weakness in a computing a communication subsystem that can be exploited with malicious intention to cause errors and failures

Threat a potential attack on a system

Attack an actual malicious attempt to exploit a vulnerability of a system

Intrusion a successful attack resulting in an erroneous state. A big part of classical security work is to prevent intrusions. An emerging field of research called *survivability* is looking at ways that systems can tolerate intrusions.

Fig 16.1 attack+vulnerability -> intrusion -> error -> failure

Example intrusions: credit-card fraud; theft of your homework from your files; identity theft; theft of cell-phone services; denial-of-service due to SYN-flooding in networks; malicious insertion of bad credit information into your credit file;...

Assessing threats and vulnerabilities: if you follow computer news sources you know that new computer viruses are being produced all the time. Some of these new viruses are more serious than others – they represent a greater threat either because of their ability to propagate quickly, to do major damage to a system's state, or to consume communication resources (or all of the above). These assessments are assessment of a degree of threat. The national threat status (red, orange, yellow,...) is a similar assessment.

Similarly, some systems are more vulnerable than others because they have more latent faults, or more serious latent faults.

F(Threat,vulnerability) = Risk – multidimensional quantity

It is interesting to observe that in the world at large, threat and vulnerability are not independent: people seem to like to attack systems that are known to have lots of vulnerabilities. Other people like to attack systems that are claimed to have few vulnerabilities.

System properties related to security

confidentiality protection of information from unauthorized disclosure: mechanisms – physical isolation; encryption (what threats remain?); authentication and authorization of access

integrity protection of information from unauthorized (or undetected) change; mechanisms – cryptographic checksums, redundancy, isolation; authentication and authorization of access.

availability accessibility of a system (or information) to its authorized users; the opposite of denial-of-service; mechanisms – isolation from threats (difficult if the goal is widespread availability)

authenticity maybe should not be included; might be interpreted as integrity of meta-data about an object

How secure are we now?

A question of risk.

- Physical vulnerability + threat to our selves
- Vulnerability of shelter, transportation, energy, food, air, water supplies + threat to same
- Vulnerability of our financial systems + threat to same
- Vulnerability of our computational infrastructure + threats to same

Unfortunately, most of these are interdependent, and the risk in each contributes to risk in the others. Example: 100M refrigerators with buffer-overflow vulnerabilities, attached to the internet, potentially constitute a powerful threat to some critical infrastructures.

Difficult to assess the overall risk as either individuals or society. Many vulnerabilities are known, some with severe consequences if exploited. It is also clear that there are enough people with malicious intent that the threat is significant but unquantifiable. The barriers to exploitation of the vulnerabilities by these people are also not well understood.

The result: people of good will disagree about the overall risk level. Personally, I don't feel very threatened on the first two points, more so on the third, and I feel at greatest risk on the last.

Next time: security mechanisms and techniques in distributed systems - in depth look at Kerberos