CSCE 742 - Practice Final
Name:

This is a 150-minute exam. On all essay type questions, you will receive points based on the quality of the answer - not the quantity.

Make an effort to write legibly. Illegible answers will not be graded and awarded 0 points.

There are a total of 10 questions and 100 points available on the test.

Problem 1

Speculate as to why none of the architecture description languages other than UML have achieved widespread use.

Many possible answers here. Some starting points for arguments:
- difficult to create critical mass to attract tool builders
- most notations only capture a small portion of architectural concerns
- Lack of communication between "academic" software engineering and commercial SE
- Too domain specific; have not evolved to describe new kinds of architectures (e.g. web services)
- Too hard to use / not understandable to regular developers

Problem 2

Create an attack tree describing how an attacker might attempt to steal money from an Automated Teller Machine (ATM).

There are, of course many many possible ATM trees. I give a partial example here.

Goal: Steal money from ATM

1. *Physical attack*
    - a. *Break ATM casing and steal money*
        - i. *[AND]*
            1. *Steal entire ATM for later dismantling*
            2. *Procure vehicle capable of transporting ATM*
    - b. *Card-data stealing attack*
        - i. *Card-based attack using new cards*
            1. *[AND]*
                - a. *Buy/steal card producing device*
                - b. *Buy/steal card stock for new ATM cards*
        - ii. *Data capture for ATM spoofing*
            1. *Capture ATM track 1&2 data and valid PIN using a skimming device*

        2. *Buy data and PINs from black market*
2. *Capture/guess ATM data from online banking site*
3. *Capture/guess ATM data through ATM software vulnerability*
4. *Get valid card in someone else's name (id theft)*
5. *...*

If you create an attack tree over multiple pages with things I have not thought about then I will get nervous and ask you about your past employment …

Problem 3

What difficulties do cyclic component dependencies lead to in an architecture? What can be done to break cyclic dependencies?

Sets of components that contain circular dependencies are much more difficult to test and maintain than components that do not have circular dependencies. They are tightly coupled, they each depend on the interface of the others. For testing, a usual strategy is to use "leveling" to test. That is, you test components that have no dependencies first, then once you have confidence in them, you test components that depend only on those components, etc. If you have circular dependencies, then it is much more difficult to test in this way, and we have to examine the behavior of all of these components simultaneously.

Similarly, for maintenance, it is often difficult to modify one of the cyclically connected components without changing all of the components in the cycle. This can be especially problematic if the components containing the cyclic references reside in multiple packages. For versioning, if components mutually depend then they must be installed and updated simultaneously.

The same thing is true for compilation. A change in one triggers compilation of all others.

Breaking cyclic dependencies:

Refactoring: using interfaces and patterns to break dependencies. Abstract interface pattern, Façade, Dependency inversion, observer, mediator, are all patterns that can break dependencies.
Combining components: combining mutually dependent components into a single component

Problem 4

We have described RPCs and messaging schemes as inter-process communications mechanisms. Suppose we restrict our attention to one process with multiple threads.

1. Are there analogous concepts to RPCs and messaging between threads?

Messaging concepts exist, but they are "manufactured" from lower level concepts such as semaphores. Event queues are used for threads as the analogue for messaging *between threads. Threads communicate using these queues (which are themselves objects that are controlled by monitors or semaphores). Delivery is guaranteed because it is within process, as long as the process does not crash.*

There is no immediately analogous concept to RPCs (except in the language Ada, which supports rendezvous). You could argue that local procedure calls act like RPCs; they are synchronous, after all. However, they do not really involve other threads at all.

2. Describe an additional means of communication that is available between threads.

Shared memory
Monitors
Semaphores
…etc…etc…

3. Do the same benefits/drawbacks between RPCs and messaging schemes exist when considering interthread communication as interprocess communication?

It is questionable whether RPCs would be useful between threads; since the threads share an address space, why not just use the active thread to perform the task in question? In addition, RPCs are synchronous, so there is no performance benefit, just additional overhead.
Messaging is still useful, however. It allows decoupling of long running processes from (for example) the user interface. Also, it allows a clean interface between communicating threads that prevents many of the problems with deadlock and race conditions.

On the one hand, you don't get the same guarantees in terms of persistence that you get with IPC. On the other, you don't have to worry about process lifecycle issues; if an application is running, then in all probability, its threads will be running too.

In terms of performance, communication will be much faster between threads than between processes, so the cost of splitting up tasks is lower.

In terms of isolation, threads are not isolated from one another, so some of the benefits with IPC do not accrue with inter-thread communication.

Also, IPC allows processes to be spread across multiple machines potentially increasing scalability, where as ITC does not.

Problem 5

What are the benefits and drawbacks of using XML vs. binary protocols for messaging between processes?

On the plus side for a custom binary protocol, a custom binary protocol may require significantly less network bandwidth than XML, and, if processes agree on the format, require significantly less translation and parsing effort at the network boundaries. For applications that are network-intensive, and whose protocols do not require significant extensibility this may correspond to significantly better application performance and network utilization.

On the other hand, XML may be much easier to maintain because it is possible to use standard tools to snoop the message traffic and diagnose application errors. In addition, it supports transparent re-hosting, even if the new platform has different endian-ness, for example. Also, it may actually lead to better performance given relatively static data because it is possible to easily insert pre-built caching applications into the network stream. XML is more likely to offer extensibility along multiple axes: it is fairly straightforward to add additional data to XML messages without disrupting existing clients, and there are several tools for manipulating and routing XML messages that could allow new applications to be easily integrated into the system. Also, many languages already have built-in support to generate and parse XML messages, so implementation might be easier than using a binary protocol.

Problem 6

Big Bang, Parallel Run, and *Staged Migration* are all techniques for upgrading existing software installations.

1. **Briefly** explain each technique:

Big Bang: Pick a day, turn the old system off and the new system on
Parallel Run: Choose a period in which the old and new systems run in parallel
Staged Migration: Swap out pieces of the old system with pieces of the new system over an installation period or migrate portions of the organization over a period.

2. Describe advantages and disadvantages for each technique with respect to the others in terms of data migration, complexity, and rollback in case of failure.

Big bang:
data migration: one shot. May not work for continuous operation system.
Complexity: simple in the sense of only one system running at a time. Lots of complexity if systems do not have downtime – how do you do "immediate switchover"?
Rollback: often difficult to recover from failure, may require reverse data migration

Parallel Run:

data migration: can be in parallel with old system running (new system will fail on certain queries so route to old system). Less risk. Need to have support for continuous migration as data is added to new system.
complexity: more complex than big bang in that policies have to be created for synchronizing and routing traffic between old and new system. How is new system validated? Voting? Split stream traffic? Also, need to maintain redundant copies of information
Rollback: can run in lockstep, in which case rollback is almost trivial. Can also run split stream traffic, in which case some capability of reverse migration of data is required.

Staged
data migration: need facades to allow new system component to work with old system components. These must translate back and forth.
complexity: Need facades for boundaries. If migrating parts of organization, may need policies for discrepancies between old/new system.

3. Describe installation scenarios where you might use each technique. Do not use examples from class.

Big bang: Home web server upgrade
Parallel run: Banking transaction software
Staged migration: Air traffic control

Problem 7

What are the **availability** benefits and risks associated with the following architectural styles: Pipe and Filter, Repository, Event-based, Layered.

In your answers, consider how each technique might be used to construct highly-available systems and what failure modes might be introduced.

Pipe and Filter: for availability: you can duplicate streams of data to multiple boxes transparently (from the perspective of the filters); if failure occurs, you redirect to good stream. Also, you can easily introduce "voters" to check for discrepancies between results from multiple filters. This is perhaps the simplest architecture to make highly available. Risks: Along a single pipe-and-filter chain, any single failure will likely cause the whole system to fail, because the filters do not "know" about each other.

Repository: this pattern can be risky for availability; it is a central store so problems in terms of consistency in the presence of multiple writers and readers/writers must be addressed. In addition, it is a central point of failure. On the other hand, it consolidates critical data in a single location, so supports single-point logging and recovery. Also, there are many schemes for making highly available repositories (e.g. database clusters). These tend to be technologically sophisticated (and expensive) but tend to work well in practice.

Event-based: This depends somewhat on the implementation of the pattern. If there is a centralized event broker that routes events between components, this can be risky for *availability (as well as performance) because it introduces a central point of failure. Also, it can be difficult to understand the composite behavior of event-based systems; "event storms" can occur in which one event leads to a cascade of many events that can reduce system reliability and availability. On the other hand, the mechanism for failover when constructing highly-available systems is usually event-based. A heartbeat event, sent at regular intervals, is the means by which system health is monitored. If a sibling system does not send a heartbeat, then failover is performed.*

Layered: There are many possible answers here. In general, layering helps construct highly available systems because it limits the kinds of failures that must be accounted for. For example, one of the reasons that the web is reliable is because of the isolated handling of classes of failures by different layers: IP handles routing, TCP handles packet retransmission in case of loss, load balancing servers handle compute resource failures, etc. etc.

Problem 8

Write the guarantees for the following AADL component describing a dishwasher mode controller:

```
system Dishwasher_Mode
	features
		door_closed: in data port Base_Types::Boolean;
		time_remaining: in data port Base_Types::Integer;
		pump_on: out data port Base_Types::Boolean;
		dishwasher_mode: out data port Base_Types::Integer;

	annex agree {**
		const SETUP_MODE : int = 0;
		const WASHING_MODE : int = 1;
		const RINSE_MODE : int = 2;
		const DRYING_MODE : int = 3;

		guarantee "the pump shall be off if the door is open" : true;
		guarantee "If the dishwasher was in WASHING_MODE and time 
remaining is zero, it shall enter RINSE_MODE" : true;
		guarantee "The dishwasher shall never transition directly from 
WASHING_MODE to DRYING_MODE" : true;
		guarantee "The dishwasher shall start in SETUP_MODE" : true;
	**};
end Dishwasher_Mode;
```

Guarantee 1: pump_on => door_closed;
Guarantee 1 (alternate): (not door_closed) => (not pump_on);
Guarantee 2: true ->
((pre(dishwasher_mode) = WASHING_MODE and time_remaining = 0) => dishwasher_mode = RINSE_MODE)
Guarantee 3: true ->
(pre(dishwasher_mode) = WASHING_MODE => (not dishwasher_mode = DRYING_MODE))
Guarantee 4: dishwasher_mode = SETUP_MODE -> true.

Problem 9

On performance.

1. What is the difference between response time and throughput?

Response time is from the client's perspective: how long does it take to service my request? Throughput is from the server's perspective. How many requests can be processed in a given time period?

2. Give an example of a system with excellent throughput but poor response time and vice versa.

*A pipelined system may involve several processors working in tandem to solve a particular problem. It may therefore be able to process very large volumes of transactions (high throughput) due to partitioning the problem into segments that are handled sequentially, while still exhibiting poor response time (each segment takes time t, with number of segments s, so the total response time is n*s).*

Instead, imagine a single processor non-pipelined system that processes requests sequentially. If there are few requests, it will have better response time than the pipelined system because there is no latency in servicing the request. However, it will have very poor throughput under heavy load.

Problem 10

What is the distinguishing characteristic of a *real time,* as opposed to a *non-real time* system? What is the difference between hard and soft real time systems?

Answer: for real time systems, an operation's correctness depends not only on its logical correctness but also in the time required to compute it.

In a hard real-time system, computation of an answer after its deadline is considered failure. Soft real time systems can tolerate missed deadlines as long as there is a bound on the number of missed deadlines within some time scale.