Final Questions, Answers, and Grading

(**1**) [5 points; = 1 point for each of (*a*)–(*e*)] What do the following terms mean for sensor networks?

These are all terms from the reading on sensor networks, pervasive, and proactive computing.

(*a*) smart dust *The best answer states that the dust "motes" have sensing capabilities, communication (either passive or active), computation abilities, and they are really tiny, itsy-bitsy in size.*

(*b*) attribute-based addressing (or attributed-based naming) *The best answer indicates that sensor nodes can be addressed (queried, sent messages, etc) by some environmental attribute, such as temperature, current location, and so on.*

(*c*) directed diffusion *This refers to a specific protocol where some nodes ask for sensor data, and a dynamic routing algorithm to return such data to interested parties in a wireless sensor network.*

(*d*) localization *This generally refers to the problem of sensors learning their position (location) in some coordinate framework, so that sensor data can be associated with a point in space; localization also refers to the service (process) of a sensor attaining it location.*

(*e*) ad hoc networks *This is generally a type of unplanned or dynamic type of wireless network that has to set up all the networking functions without the benefit of a static routing or forwarding specification.*

(**2**) [3 points] Generally speaking, for Remote Procedure Call, which style will use smaller-sized messages, stateful or stateless? *Stateful systems can allow smaller messages because RPC messages do not need to have so many parameters (instead, the same information is kept as part of the "state" for a caller tracked by the server). Several students showed they knew this, but claimed this implied stateless methods would have smaller messages; some partial credit was given in these cases.*

(**3**) [2 points] What is a causality error in a simulation? *When a simulator's next scheduled event's firing time is prior to the current simulation time, we have a causality error (events processed out of order).*

(**4**) [5 points, one point for each of (*a*)–(*e*)] Which of the following are idempotent operations?

(*a*) Lookup the telephone number of Acme Incorporated.

(*b*) Assign the telephone number of Acme Incorporated to be 1-800-ACME-INC, regardless of whatever it was before.

(*c*) Delete the file named `mydata.txt`.

(*d*) Request a write-lock on database item `X`.

(*e*) If `A` is positive, assign $A = A - 1$, otherwise do nothing to `A`.

Clearly (a) and (b) always have the same result if repeated, but (e) may not. As for (c) and (d), it depends on the system semantics – several students explained this. In Unix, (c) is idempotent, but other systems could throw an exception for trying to delete a missing file. Idempotence of (d) depends on whether a system will complain, do nothing, or block if the lock owner tries to write-lock an item that it already owns. A few students confused idempotent with non-idempotent, and this was obvious from their explanations (they got partial credit).

(**5**) [5 points] Which of the two types of network protocols, UDP or TCP, would likely be better suited to RPC methods with "call exactly once" semantics? *Because "call exactly once" needs some kind of guarantee, and TCP guarantees in-order delivery of bytes, with each byte delivered once, TCP is the best answer.*

(**6**) [10 points] What are the two queue invariants that can guarantee each simulator in a networked simulation always processes events in the correct order? *First, the queues should be ordered by timestamp (and one could also remark that each simulator should buffer its messages to be sent, until it is certain that no earlier message will ever be generated before actually sending a message). Second, a simulator should wait until all its input queues are non-empty (the only exception to this would be a guaranteed lookahead value for a specific system).*

(**7**) [5 points] Explain why RFID tags could be used for more than just business inventory applications (that is, more than just tracking items being shipped, more than totaling prices and quantities of items being purchased in stores, and so forth). *The best answer mentions that the new, high-end generation of RFID tags (this was amply described in the reading material) have storage capabilities, some have batteries, and sensors, and these can enable all kinds of new applications. Several students mentioned that RFID tags could be used to track people in building, pets, and so on – but this is actually just an extended inventory application, nothing really new; some students said that with fine-grained inventory, enabled by RFID, businesses could better predict trends, but again, this is just an extension of the inventory application. A few students mention that tracking RFID tags can be used to solve some kinds of co-location problems, such as when two entities come near each other: partial credit for this (not the best answer).*

(**8**) [5 points] The head programmer at Acme Incorporated has been asked to devise some way to check for situations when the networked database transactions get into a deadlock situation. She works hard to get a correct program, one that never aborts transactions needlessly: in fact, she has a mathematical proof that her algorithm will never claim that a set of transactions is deadlocked, when in fact no deadlock situation exists. Yet, when her program is run, the users complain that her deadlock detection algorithm is faulty. How can this be? *The problem with the algorithm is that it misses detecting some deadlocks; this point was emphasized in lectures, that we need two properties for correctness, a guarantee of finding current deadlocks and the lack of phantom deadlocks.*

(**9**) [5 points] Explain why it could be a bad idea for the implementation of RPC to retry sending a server request if no response is received after a timeout period. *If the operation is not idempotent (such as ordering a book, when a new request will order another book) then we have a bad (unintended) result. A couple of students said that it would be a problem for real-time applications, but it's difficult to explain why this will produce a bad situation (after all, if the application was real-time to begin with, it would not be resending after a timeout). Many students cited that resending would be bad in case of a deadlock, server crash, etc — so the only bad thing would be consumption of resources. Minor partial credit for that point.*

(**10**) [3 points] One idea for sensor networks could be to use all the Internet protocols (IP, UDP, TCP, HTTP) for communication between sensor devices and between gateways and sensor devices. Give up to three different reasons why using the Internet protocols may not be well suited for sensor networks. *Several ideas come to mind: IP addressing is not suited to sensor networks; IP packet size and processing overhead is likely too much for some sensor networks; sensor networks are not only wireless (IP can be wireless too, as in WiFi networking), but nodes can be off to save energy, or mobile; sensor networks also have the potential of in-network processing, which allows them to aggregate data in the network, something we can't usually do with Internet protocols – we need some new features beyond what the Internet protocols offer.*

(**11**) [2 points] Two general system types can be simulated by computers, *open systems* and *closed systems*. Open systems model situations like customers that arrive to a restaurant (the restaurant is the "system", and clients come from outside the system) and leave after being served. Closed systems model situations where the entities in the system are fixed, such as isolated computer networks with a fixed number of users. We saw how the exponential distribution could be useful to model an open system, since that distribution approximates the way customers arrive over time. Explain why the exponential distribution is also useful for modeling closed systems. *The service times of units in closed system can be modeled by the exponential distribution, just as service times are modeled in open systems.*

(**12**) [5 points] In Java and Python, a socket is an object with methods for sending and receiving messages. Could a socket object be passed as a parameter to a remote procedure call? If not, why not? If your answer is yes, what would be the marshaling and demarshaling? *The simple answer is no: standard RPC does not deal with complex objects, only with simple variables and structures. However, even if RPC could deal with more complex objects, how could it handle a socket that is the programming abstraction of a connection between two points in a network? How could the method being invoked, say as host B, use a socket that only has meaning as a connection between host A and host C? Perhaps there are a few possible cases where this could have meaning (a closed socket, a pattern socket, an RPC that just does a few trivial operations with a socket) — but the general answer is no: a socket cannot be passed as a parameter to a remote procedure.*

(**13**) [5 points] What technique is used to describe parameters to a remote procedure call in a language-independent way, so that the same RPC can be invoked by different languages such as C, C++, Java, etc., and stubs for these languages can be automatically generated? *IDL – Interface Description Language. The answer "marshalling and demarshalling" is incorrect, because those are procedures, not description techniques (you need a description of parameters first, before the marshalling and demarshalling procedures are composed).*

(**14**) [6 points] What are possible advantages of using a *coordinated group* of inexpensive, but lower quality sensors rather than using one expensive, high quality sensor to do the same task? (More points to answers that refer to issues from the reading material, fewer points to general "common sense" answers.) *Numerous points include:*

- *greater spatial coverage by numerous sensors (including odd shapes, various densities, and so on)*
- *fault tolerance — no single point of failure; in fact, if sensors are so inexpensive as to be "disposable", then they can be manufactured and deployed with minimal quality control, because we can plan on a certain fraction of them being faulty*
- *the duty cycle of numerous sensors can be rotated so that some are sensing while others are recharging (from solar power or other similar techniques)*
- *the placement needs for many sensors can be less careful than for a few, expensive ones: we can have nearly random deployment*
- *the paper (by Estrin et al) shows how multiple sensors can triangulate a sensed object, and there are other similar benefits you get with multiple vantages*
- *routing can be lower power (energy cost) in a dense sensor network than for a sparse wireless network*
- *the redundancy of multiple sensors can be exploited for calibration, better trust of the average sensor reading, and so on.*

Note: the answer "because it is cheaper" is incorrect: it could be that the combined cost of many inexpensive sensors sums to be about the same as the cost of one, expensive sensor.

(**15**) [5 points] How can a deadlock situation arise in a networked simulation? *The answer is that two (or more) simulators can have all at least one input queue empty, so in effect each simulator is waiting for an event from the other – "null events" were invented to take care of this problem. Note that unlike databases, where deadlock is due to a wait-for cycle of resource locking, the deadlock of simulation is a communication deadlock.*

(**16**) [3 points] Suppose a distributed system has hardware clocks in each computer that are synchronized to within one microsecond of each other (at all times, for any two hosts, their hardware clocks agree to within one microsecond). Can a networked simulation take advantage of this hardware situation? Explain why or why not. *No. Simulation time generally has nothing to do with real "wall clock" time. Even in real-time simulated interactive games, the simulation will have its own internal clock, apart from the computer hardware clock. One student answered that the synchronized hardware clocks could be useful for timeouts (needed to resend messages); a little credit for this: TCP for instance does use hardware timers, but they don't need to be synchronized. Perhaps the tuning of when null messages are sent could be improved by synchronized clocks, but this isn't obvious. The idea of scaling simulation time to the hardware clocks (say, by a linear transformation from hardware time to simulation time) must take into account the delays*

of processing events, message transmission and propagation, and so on. So the approach of using the hardware clock to assist time-stepped simulation needs more assumptions about time than just hardware clock synchronization.

(**17**) [5 points] Parallel and distributed event-driven simulators use messages to communicate events between simulators at different hosts. What could go wrong if each "message send" were replaced by an RPC call that adds an event to the list of events at another host's simulator? *If an RPC call is allowed to directly deposit a generated event on another simulator's event list, then that simulator will be free to fire that new event regardless of whether or not some future event could be generated with an earlier simulation time: the result is that we can have a causality error (events processed out of order).*

(**18**) [6 points, 2 points for each of (*a*)–(*c*)] The concept of a transaction in a database is very general; transactions can read and write any number of items before they commit. Some systems don't allow such generality: transactions are restricted to be of certain types in these systems. Call a system *Type I* if any transaction is either (*i*) a transaction that only reads items, or (*ii*) a transaction that reads no items and writes to exactly one item (and it can only write *once* to that item). Call a system *Type II* if any transaction is either (*i*) a transaction that only reads items, or (*ii*) a transaction that reads no items and only writes to items — it can write to several items, but to each item, it only writes once. Call a system *Type III* if every transaction just writes items (again, to any item, a transaction will write to it at most once); a Type III transaction can write to several items.

(*a*) Is two-phase locking needed for a Type I system?

(*b*) Is two-phase locking needed for a Type II system?

(*c*) Is two-phase locking needed for a Type III system?

For each of these, the fundamental question will be: is the result of running transactions concurrently the same as executing them in some serial order? For (a) the answer is, from the point of view of the database, yes, even without locking (2PL or otherwise); however, if you also insist on query integrity, the answer is no — because a query could return a result from some inconsistent view of the database. So the answer for (a) can either be yes or no, depending on how you justify your answer. Part (b) does require locking, because the result of running two update transactions concurrently could produce a database not equal to any serial schedule; the same is true of (c).

(**19**) [5 points] Transactions can have complex logic: a particular transaction could be programmed to sometimes write to item `X`, sometimes write to item `Y`, or perhaps write to neither of these, depending on what value another item `Z` has when the transaction runs. Call a transaction *simple* if the items it writes are the same set of items in any execution of the transaction (also, a simple transaction always reads the same set of items in any execution). Suggest how the problem of managing concurrent transactions in a distributed database could be easier to solve (that is, the coordination algorithm could be less complicated) if all transactions were simple. *Simple transactions would allow a system to have the knowledge, at the beginning of a transaction, of exactly what items will be read and written. This knowledge could*

be expoited by scheduling transactions based on intersection of read and write sets, or by locking all the variables immediately, at the start of the transaction: if they can all be obtained, the transaction goes ahead, runs to completion, commits, and the locks are released. This simplifies the distributed processing.

(**20**) [10 points] Some database transaction managers can be tuned to give priority to some transactions in the case of blocking: a priority transaction, when it requests a lock held by a non-priority transaction, causes the non-priority transaction to abort immediately. This idea has to be changed for a distributed database using two-phase commit: a non-priority transaction cannot be aborted if it is in the "prepared to commit" (uncertain) state. But can such a non-priority transaction be aborted if it has not yet reached the "prepared to commit" state? That is, will database consistency (ACID) properties still hold over the distributed database, if priority transactions are allowed to abort any blocking non-priority transaction that has not yet reached the prepared state? Explain your answer in terms of messages and steps of the transaction coordinator. *This was meant to be an easy question. Since two-phase commit does preserve the ACID properties, the subcase where some transactions are aborted (by priority) still has the ACID-preserving qualities of two-phase commit. Only one or two students bothered to answer in terms of messages and coordinator steps, which did help in scoring a correct answer.*