Question 1 (25 points)

a) When is it acceptable to use eventual consistency rather than strong consistency?

- Network partitions heal within a short time
- Conflicting writes are few/rare.
- Impact of out-of order data or stale data is negligible.
- Availability dominates consistency / one-copy semantics

(Answers may be variations/improvisations on this. Answer may also involve some real example or scenario where the model is applicable)

b) What safety properties does Paxos strive to enforce?

- A majority of nodes agree on the same value
- In any round, at most one value is ultimately chosen
- The chosen value was proposed by some node
- A node is never told that a value is the consensus value unless a majority has agreed on that value.

c) Two-phase commit is prone to indefinite blocking because of coordinator failure. How does Paxos address this problem? What does it sacrifice in return?

Paxos addresses this problem by allowing any node to spontaneously become a proposer. It sacrifices unanimity, a much stronger property than majority. Also, termination is not guaranteed, and could result in wasted processing and network resources. Paxos does *not guarantee* liveness.

d) How does Paxos handle the situation where there are multiple concurrent proposers that are unaware of each other?

Paxos lets acceptors accept multiple proposals. However, it enforces a total ordering on the proposals. The acceptor only accepts proposals whose proposal number is strictly greater than the highest numbered one it has already accepted.
Additionally, proposers gather any values already written to the acceptors, and are required to propose the same value as the highest proposal that acquired majority so far, to help ensure a consensus value is reached even if multiple proposals occur.

e) In Paxos, it is essential that acceptors never recant. In other words, an acceptor never changes the value v ($\equiv a_k$) after it is written to its stable storage. Explain why this is important.

In Piazza we clarified this problem a little bit:
A couple of you have brought up the fact that it is possible for a proposer to assemble a majority, but only manage to get less than a majority of acceptors to write the value Ak (e.g., due to network faults). A second proposer can then assemble a majority (and yet never see the value Ak), and can therefore request a different value of Ak to be written. Depending on timing, a particular acceptor may see the write request from both proposers. Since the second proposer has a higher id n, the acceptor will overwrite the Ak value of from the first proposer with the one from the second.

This is allowed by the protocol. Until a majority of the acceptors have written a value Ak, and outstanding proposers have completed, it is still subject to change, but only for strictly higher id proposers. Likewise, the fact that a value was written should not be forgotten.

So the "definition" of recantation in problem 1E is a bit loose -- really, it isn't that the value can never change, but rather that the value is not changed except as defined by the protocol, and is not forgotten.

If acceptors could recant (i.e., change or forget what is accepted), then the system cannot converge on a majority consensus. Even if it does, the acceptors contributing to that majority may change their minds, and the consensus disappears. Given the clarified definition, recantation can be seen as a Byzantine fault, and is therefore outside the scope of Paxos.

Question 2 (20 points)

A)

This is a name/name conflict.

In practice, today's Linux systems name core files as "core.<pid>". So the chance of having a conflict will be very low.

B)

This is a remove/update conflict.

In such a case, Jane can choose to accept all the files generated by John's make.

Question 3 (25 points)

Optimistic, for eventual consistency.

Networks are likely to be flakey at construction sites. However, at least read access and possibly update access to the blueprints will be needed. So availability dominates as a design consideration. Since usually users will update different parts of the same file, optimistic replication is okay, and unmergeable updates will be rare.

Pessimistic decreases availability because coordinating replicas usually requires locks to do write updates to file contents (reads can be blocked during coordinated update propagation as well). Network latency (and esp. on mobile devices unreliability of network) further slows pessimistic replication down because of locking for updates.

Question 4 (15 points)

In the context of pessimistic replica control of byte ranges within files (not whole files), answer the following questions.

A. Give a use case in which read-one, write-all is a reasonable strategy.

Many possible answers here. Some acceptable answers are:

- We're interested in low latency reads. We only need to read the response from one server.
- We're interested in high availability of reads. All of the servers have to go down before we can stop reading.
- Reads are much more frequent than writes.

B. Give a use case in which write-one, read-all is a reasonable strategy.

Given the pessimistic replica control specification, there are not a lot of good answers here. The cases that make sense are things that are never modified once written, or append-only logs, where the order of the entries is not critical. See the discussion of Enqueue and Dequeue in the optional reading by Herlihy.

- We don't care for high availability of the most recent data, we're interested mostly on having fast writes.
- Writes are much more frequent than reads.

C. Suppose your boss has mandated use of a read-one, write-all strategy for all deployments. In the workload of the particular customer you are working with, writes are much more frequent than reads. You have 3 sets of servers that you can deploy as replicas. Choice-1 has one server with a read/write latency of 100 ms, and two servers with read/write latency of 400 ms. Choice-2 has one server with read/write latency of 100 ms, one server with read/write latency of 225 ms, and one server with read/write latency of 250 ms. Choice-3 has three servers with read/write latency of 225ms.

Which set of servers would you use for this customer? Justify your answer.

The best option is Choice-3. With Choice-1, we have to wait 400ms for a write, since we have a write-all strategy. Reads would be very fast with Choice-1 (100 ms), but in this use case writes are much more frequent than reads. The same logic applies for Choice-2, the write latency would be 250ms. Choice-3 offers the lowest write latency, 225ms.

Question 5 (15 points)

Assume each replica has one vote.

A. 5 replicas, maximum read availability
For maximum read availability, we will have r = 1.
We must have r + w > 5, thus w > 4, thus w = 5.

B. 8 replicas, ability to update with at least two replicas up
This is not possible as for 8 replicas, we must have w > (8/2) = 4.
This assumed each replica has the same amount of votes.

C. 8 replicas, ability to update with at least five replicas up
w = 5

we must have $r + w > 8$, thus $r > 3$, thus $r = 4$.