Failure Detectors

Today's papers, presented by Tina Nolte, tnolte@mit.edu:
*** Chandra, Toueg, Unreliable Failure Detectors for Reliable Distributed Systems
*** Aguilera, Chen, Toueg, Using the Heartbeat Failure Detector for Quiescent Reliable Communication and Consensus in Partitionable Networks
*** Gupta, Chandra, Goldszmidt, On Scalable and Efficient Distributed Failure Detectors

For next lecture, read:
*** Hadzilacos and Toueg, §1-5.5, 7, 9
*** Birman et al., §1-4 (skim, with emphasis on §4)

1 Introduction

The asynchronous model of computing with process failures comes with it a large number of impossibility results. While we know that some problems such as consensus are not solvable in an asynchronous system with faults, we do know that the problem is still important. Today we'll discuss failure detectors, one tool for overcoming some of these problems. We'll start by briefly describing some properties of failure detectors and how they might be used and then describe the use of a particular kind of failure detector called heartbeat that can perform work in partitionable networks. Afterwards, we'll study some ideas on how to talk about quality of service and scalability of failure detectors as well as a configurable algorithm that satisfies some of these requirements.

2 Unreliable Failure Detectors

We will start by considering an asynchronous communication model with reliable links and processes that can fail by crashing only (no recovery). Later, we'll be relaxing some of these restrictions. Every process has access to a local failure detector module. These modules monitor sets of processes in the system and reports who it currently suspects to have crashed. These modules are permitted to make mistakes by either suspecting a correct process of crashing or by not suspecting a process that has crashed. However, the modules can correct themselves. This could lead to, for example, a module repeatedly adding and removing processes from its list of suspects. Also, modules for different processes can have different lists of suspects.

Now we'll define two words that often occur in the literature of failure detectors: completeness and accuracy. Informally, completeness requires a failure detector to eventually suspect every process that actually crashes. Accuracy restricts the mistakes a failure detector can make in suspecting correct processes.

2.1 Failure detector properties

Section 2.3

The first paper describes 2 completeness and 4 accuracy properties. For completeness, we have:

Strong Completeness: Eventually every process that crashes is permanently suspected by every correct process.

Weak Completeness: Eventually every process that crashes is permanently suspected by some correct process.

For accuracy we have:

Strong Accuracy: No process is suspected before it crashes.

Weak Accuracy: Some correct process is never suspected.

However, even weak accuracy can be a tall order. As a result, we introduce two more accuracy properties:

Eventually strong and eventually weak accuracy. These basically mean that there is a time after which strong accuracy or weak accuracy holds.

Section 2.4, 3

These 2 completeness and 4 accuracy properties describe 8 basic classes of failure detectors. However, a very simple reduction shows that every weakly complete failure detector class is actually equivalent to its strongly complete correspondent. Each process maintains its own output variable which basically will be the output of the stronger failure detector whenever the process queries it. The reduction simply gets each process to periodically send its local failure detector's list of suspects to all processes. When a process receives one of these suspect lists it adds all of those suspects to its output list, and removes the process that sent it from that list. This is returned as its list of suspects. This will ensure that if some correct process permanently suspects a crashed process then eventually all correct processes will permanently suspect it. You also have to check that accuracy properties are preserved, but these arguments are not difficult and I refer you to the paper if these seem confusing.

The paper goes through solving consensus using the classes of failure detectors. It turns out that the weakest failure detector in these classes (there is a time after which some correct process is never suspected by any correct process and eventually every process that crashes is permanently suspected by some correct process) is still usable to solve consensus in systems where the majority of processes are correct. I'll refer you to the paper for details.

2.2 Warmup: Using a Strong Failure Detector to Solve Consensus

Section 6

[p. 240]

The strong failure detector satisfies both strong completeness and weak accuracy. Eventually every process that crashes is suspected and some correct process is never suspected.

Now, let's have each process i maintain a vector of values with one entry for each process index. Basically, we store the initial value of process j in position j of the vector whenever we learn what that value is. Processes also maintain a variable for the current round number.

The algorithm runs through 3 phases:

Phase 1 has processes going through n-1 rounds, broadcasting and relaying messages with its round number and vector. A process only proceeds to the next round if it receives a current round msg from every process not suspected. When a process has collected these values it fills in any blank entries in its value vector that it can.

In phase 2, each process sends its vector to all other processes. The process waits to receive vectors from all processes that are not showing up as suspected. This time, we erase entries from our vector that were blank in any received vectors.

In phase 3 we decide the first non-empty component of the vector.

Correctness sketches:

Uniform validity (a value decided was proposed) should be obvious.

Termination should also be relatively simple since we only need that in each round, etc., processes get enough replies from other processes. However, since correct processes send messages and crashed processes eventually show up as failed this is no problem. Also, there is at least one correct process c that is never suspected, so we will not have a problem with blank vectors.

Uniform integrity (processes only decide once) should be obvious.

For agreement we need that processes get the same vector at the end of phase 2.

Consider phase 1. In a round before n-1 we know that all non-crashed processes wait for a correct process c. Hence, it is clear that each process incorporates this information into its vector.

In round n-1, if c receives a value for index q for the first time, we know that the value must have been relayed through a chain of all processes. So, again, the value stored by c will be in other processes' vectors. This allows us to conclude that all noncrashed processes have vectors that at least contain the information that c's vector contains at the end of phase 1.

This means that in phase 2 when all noncrashed processes wait for and receive c's vector, they will trim their vectors to be the same as c's, giving us that all noncrashed processes get the same vector at the end of phase 2.

3 Heartbeat Failure Detectors

3.1 Properties of Heartbeat Failure Detectors

Section 5.2

It turns out that the use of failure detectors can be extended in a number of ways, probably not surprisingly. For example, there are papers concerned with the use of failure detectors in system models allowing not just process crashes, but also recoveries. Right now we'll examine the problem of using failure detectors in systems with process crashes (no recoveries), fair links (where a message sent an infinite number of times is eventually received, messages are not received more times than they are sent, and a crashed link eventually stops transmitting messages), and partitionable networks. Some of the definitions need to be retooled just a tad to talk about this. We modify the completeness and accuracy properties to hold within partitions. Note that processes that crash are in their own partition.

In particular, the eventually strong failure detector for partitionable networks becomes:

Strong Completeness: For every partition P, there is a time after which every process not in P is permanently suspected by every process in P.

Eventual Weak Accuracy: For every partition P, there is a time after which some process in P is permanently trusted by every process in P.

3.2 Solving Consensus using an Eventually Strong FD

Section 5.4

We can weaken the eventually strong failure detector even further by changing these definitions from holding for every partition to simply holding for the largest (if more than one largest partition exists, just select the one containing the highest process ID between all the largest ones). This failure detector can be used in partitionable networks to solve consensus if we assume the use of certain communication primitives.

3.2.1 Communication Primitives

Let's go over the properties of these primitives. We'll talk about their implementations later:

(*) Reliable Broadcast consists of 2 primitives: broadcast(m) and deliver(m). Every broadcast message contains the identity of its sender and a sequence number, making the message unique. The primitives satisfy the following properties:

Validity: If a correct process broadcasts message m, then it eventually delivers m.

Agreement: If a correct process p delivers a message m, then all processes in the partition of p eventually deliver m.

Uniform Integrity: For every message m, every process delivers m at most once, and only if m was previously broadcast by the process indicated in the message header.

Partition Integrity: If a process q delivers an infinite number of messages broadcast by process p, then q is reachable from p.

(*) We will also use two primitives called quasi-reliable send and receive, which are used to communicate between just two processes. It satisfies the following properties. Messages can be sent more than once on the same link. This might sound odd, but it turns out that the consensus algorithm we are about to consider can have a process send the same message more than once in different rounds:

Quasi No Loss: For all $k \geq 1$, if processes s and r are in the same partition and s qr-sends m to r k times by time t, then r eventually qr-receives m from s at least k times

Uniform Integrity: For all $k \geq 1$, if r qr-receives m from s exactly k times by time t, then s qr-sent m to r at least k times before time t (this property with quasi no loss means that in the same partition r qr-receives m from s as many times as s qr-sent it. However, do note that the property of uniform integrity actually gives us something more, namely that if a process qr-sends to somebody not in the same partition, then that process only qr-receives it at most the number of times it was sent.)

Partition Integrity: If r qr-receives messages from s an infinite number of times then r is reachable from s

3.2.2 The Algorithm

Ignoring the implementation of these primitives for a moment and just concerning ourselves with the guarantees they offer, let's take a look at the algorithm for solving consensus:

[page 16 of the heartbeat paper]

This algorithm uses the rotating coordinator paradigm and proceeds in asynchronous rounds. All processes are required to know who is present in the system. During some round r, the coordinator will be process c = (r mod n) + 1. All messages are either to or from the current coordinator. As coordinator, a process tries to determine a consistent decision value. If the coordinator is correct and not suspected by any surviving process then it will succeed and will broadcast this decision. In Phase 1 of the algorithm, every process sends its current estimate of the decision value, timestamped with the round number in which it adopted the estimate, to the current coordinator. In phase 2, the coordinator gathers estimates from more than half the processes, selects the one with the highest timestamp, and sends it to all the processes as their new estimate. In phase 3 there are 2 possibilities:

1) a process receives an estimate from the coordinator and sends back an ack to indicate it has adopted the estimate or

2) a process's failure detector module suspects the coordinator of being faulty. The process sends a nack back to the coordinator.

In phase 4 the coordinator simply waits for replies from more than half of the processes. If all replies are acks then c knows that a majority has adopted the estimate and the estimate is locked. It broadcasts a decision message. If a process receives such a broadcast it decides accordingly.

3.2.3 Correctness

For correctness we will be verifying familiar properties of consensus plus one or two new ones:

Uniform Integrity: Every process decides at most once.

Uniform Validity: A process can only decide a value that was previously proposed by some process.

Uniform Agreement: No two processes in the entire system decide differently.

Since we are dealing with partitions, we break the termination requirement into two separate requirements:

Partition Termination: If a process decides, then every process in its partition decides.

Termination: Assuming that the largest partition contains a majority of processes then if all processes in that partition propose a value then they all eventually decide.

The following property is a new one, concerning itself with processes being able to shut up. This can be a concern on networks where bandwidth and load can be a problem:

Quiescence: Every process eventually stops sending messages for this invocation of consensus.

Proofs:

Uniform Integrity: This follows from the code. Once a process decides it no longer executes the algorithm.

Uniform Validity: This follows from the code and the assumptions about uniform integrity of the broadcast primitives (the only received messages were sent by somebody).

Agreement:

If any process decides then it must have previously delivered a decide message, broadcast from the coordinator. This coordinator must have received ack messages from a majority of processes in phase 4. Let r be the smallest round number for which ack messages from a majority of processes are sent to a coordinator. We claim that for all rounds at or after r, if that round's coordinator sends an estimate in phase 2 then the estimate is the same as the estimate of the coordinator in round r.
The proof is an inductive one. The base case is trivial. For the inductive step, we basically just note that in phase 2 if the coordinator of a later round sends an estimate then it had to have received estimates from more than half the processes. So there is some process p that both sent an ack message to the coordinator of round r and sent an estimate message received by the coordinator for the current round (majorities intersect). The timestamp of the message in round r had to be r and since timestamps are nondecreasing we know that in the current round the timestamp is at least as large. Now let t be the largest timestamp on any estimate that the current coordinator has received. It must be at least as large as r but by the code cannot be larger than the current round. According to phase 2, the coordinator will adopt the estimate associated with this timestamp t. By the induction hypothesis, this estimate must be the same as the estimate for round r.

(Note that this is similar to Paxos, though Paxos solves a harder problem; in Paxos, multiple rounds may be occurring at the same time, with different leaders.)

Partition Termination: The result follows from agreement of reliable broadcast. (If a faulty process decides, its partition is empty so the proof would be trivial.)

Termination:

To prove termination we introduce a new definition:

We say that QuiescentDecision(P) holds if:

1) all processes in P eventually stop sending qr-send messages

2) if P contains a majority of processes and all processes in P propose a value, then all processes in P eventually decide.

Now we prove two lemmas.

Lemma 19: For every partition P, if there is a time after which no process in P qr-receives messages from processes outside P, then QuiescentDecision(P) holds.

Proof: Call the time after which no process in P qr-receives messages from processes outside P t.

To show the quiescence part of QuiescentDecision there are several cases:

Case 1: Some process in P decides. Then, by partition termination, all processes in P decide, meaning they all stop sending qr-send messages.

Case 2: No process in P decides. Then either each process is blocked by a wait statement, meaning they all eventually stop sending qr-send messages or some process in P , call it p, that proposes does not get blocked. Since p does not decide, it starts every round. Now if the partition does not contain a majority of processes then p must wait forever in Phase 2 for estimates, a contradiction. Otherwise, we know by the eventual weakness of our failure detector that there is some t' after which some process c in P is not suspected by anybody in P. Now consider a time after which this time t' has passed and the time t, when no process in P qr-receives messages from processes outside P, has also passed. Say that the largest round number at this time is r. Find the first two rounds larger than r where c is the coordinator. By assumption, p trusts c. Also we assumed that p finished Phase 3 for the second round where c is coordinator since it is not blocked and starts every round. It must have received a qr-receive with c's estimate. In order for that to have happened c had to have started the second round for which it was coordinator. This means it had to have finished its first round as coordinator. C must have received messages from a majority of processes in Phase 4 of its first round as coordinator. All processes that receive these messages from c must be in P, by assumption. C must have received messages that were acks since we assumed that it's operating after the point where all processes in the partition trust it. Hence it must have decided (by the code) and we have a contradiction.

To show the second part of QuiescentDecision, we need to show that if a partition P contains a majority of processes and all of them propose, then they all decide. Because of partition termination, we only need to show that somebody decides. For contradiction, let's say nobody does. By what was shown just above we just need that no process in P is blocked forever at a wait statement. This is done by contradiction. Pick the smallest round at which some process does block at a wait statement. Because all processes propose, then a majority of processes have just qr-sent proposals. If the coordinator is in the partition, then it will send out an estimate, every process will receive it and return an ack or nack and c completes, giving us that everybody completes, a contradiction. If the coordinator is not in the partition then we know that by strong completeness of our failure detector the processes will eventually suspect the coordinator and so do not block at the wait statement, another contradiction.

Now we want to show that for any partition, *QuiescentDecision* holds:

Proof: The paper does this by introducing a binary relation on Partitions where the relation from P to Q is true iff P and Q are distinct and there is a fair path from P to Q. To show the lemma true, we do induction on this partial order. Pick any partition P and assume that for all Q such that the relation holds from Q to P, then QuiescentDecision(Q) holds. Now we just need that QuiescentDecision(P) holds. Pick any partition Q such that the relation from Q to P holds. QuiescentDecision holds for Q and hence every process in Q eventually stops sending qr-send messages. By Uniform Integrity of qr-send and qr-receive we know that at some point no process in P qr-receives a message from the processes in Q.

If we chose a partition Q where there is no fair path from Q to P then by partition integrity of qr-send and qr-receive we know that eventually p won't qr-receive a message from Q. This allows us to conclude that eventually no process in P qr-receives messages from any other partition. By Lemma 19, we have that QuiescentDecision holds.

This allows us to finally complete the proof of termination:

Let P be the largest partition. By assumption since the largest partition contains a majority of processes we can simply apply the fact that for every partition QuiescentDecision holds.

Finally, we reason through why it is the case that the algorithm is quiescent. Each process invokes only a finite number of broadcasts in Phase 4 since they will eventually deliver their first broadcast and stop broadcasting after that delivery. Also, the lemma covering QuiescentDecision gives us that each live process only invokes a finite number of qr-sends. Assuming quiescent implementations of broadcast primitives we have that the algorithm is quiescent.

So we know that the algorithm solves consensus, but it is not trivial to implement the broadcast primitives used in the algorithm. How do we do that? Here is where the authors introduce a new kind of failure detector, one that might sound like something you've seen in practice, called heartbeat. But you might ask if there was some other way to write an algorithm using this failure detector so that we don't need to consider another failure detector...

3.3 Quiescent Consensus for Partitionable Networks Not Achievable with Only Eventually Strong FD

Section 5.3

We can actually show through a very simple argument that it is not possible to solve quiescent consensus in this network setting using only an eventually strong failure detector. This is even assuming that there are no process crashes, there are links between all processes, each link is eventually up or down, a majority of processes are in the same partition, and all processes propose a value 0 initially. The failure detector is even a little stronger than the one we just used: it's eventually strong, but not weakened to holding only for the largest partition.

Proof:

This is a proof by contradiction. Let's say we did have an algorithm A that could use the eventually strong failure detector to solve quiescent consensus. We assume a network of at least 5 processes. Now we describe 3 runs of A such that we violate the specification of consensus:

Run 0:

Let say there are 2 partitions, one with procs 1 and 2 and another with the rest. We'll say that within the partitions no messages are ever lost. Each process in a partition only trusts itself and the other processes in its partition. Procs 1 and 2 can't decide in this run. If they did, we could then have the partition merge with the other partition where those procs could have decided differently. Since A is quiescent, there is some time t_0 after which no messages are sent or received in the run.

Run 1:

Up to time t_0 this run will be identical to the first. At the next time step, the network partitions permanently become just the process 1 alone and the rest together. Again, from this time, no messages within partitions are lost and processes only trust other processes in the partition. Since A is quiescent there is a time t_1 after which no messages are sent or received in this run.

Run 2:

Now we assume a single partition with all processes. Throughout the run, process 1 and its failure detector module behave as in run 0. All other procs and their modules behave as in run 1. From time $t_0 + 1$ to t_1 all messages to and from process 1 are lost. As in the first run, process 1 will not decide. This violates termination since all processes in the partition proposed a value. When we let messages start travelling freely between process 1 and the other processes, which we will eventually do since process 1 is in the partition, it's too late. No more messages about consensus are allowed since we assumed that A was quiescent and this run looks no different from one of the prior 2 runs for all processes.

3.4 What is heartbeat?

Section 3

So we finally define a new kind of failure detector. What is a heartbeat failure detector? In general, you can think of a heartbeat failure detector as one that provides something like eventual strong completeness but only strong accuracy with probability less than 1. A heartbeat failure detector in a partitionable network provides, instead of a list of suspects, a vector of counters for each process in the system. Basically, a vector entry increases if the corresponding process is in the partition and doesn't change otherwise. A heartbeat failure detector satisfies a couple properties:

Completeness: At each correct process p, the sequence of the heartbeat values for any process q not in p's partition is bounded. This corresponds to those processes failing as far as p is concerned.

Accuracy: The heartbeat sequence of every process is nondecreasing

At each correct process, the heartbeat sequence for every process in the same partition is unbounded.

Section 6

One way to accomplish this would be for processes to periodically send a heartbeat message to its neighbours with a list containing its id attached. When a process receives a heartbeat message that does not contain its process id it appends its process id to the list of ids in the message and forwards it to its neighbours who appear at most once in the message's list. If the message already includes a process's id then the process increments its heartbeat counters for all processes whose id appears after its own in the message.

I'm not going to run through the proof that this satisfies the conditions of heartbeat, but mostly the proof is very intuitive.

A point of interest here is that this algorithm is very similar in nature to that of vector clocks. Processes maintain vectors with entries for other processes they know and increment these entries when the corresponding process is heard from under certain conditions. The purpose is different but this algorithm does seem to borrow heavily from the idea of vector clocks.

An important difference if in message size, however. The counters used by the heartbeat algorithm are unbounded, but are locally stored and never transmitted in a message. Bounded message size is an attractive feature in any algorithm.

Aside: note that while the authors are using heartbeat to implement a quiescent algorithm, heartbeat is not itself quiescent. The authors acknowledge this seeming ambiguity, and argue that it is nevertheless reasonable to have such a system, since the cost of the failure detector is amortized over the (hopefully many) processes that use it. In lecture, we surmised that it may also be possible to shut off the failure detector after an algorithm that uses it finishes executing.

3.4.1 Using heartbeat to implement communication primitives

How can heartbeat be used to implement the quiescent communication primitives? We'll start with:

Reliable Broadcast

Section 4

[Figure 2 on p. 9 here]

At each process a local variable called got is kept for each message. It stores a list of processes, intuitively the ones where the process has evidence that they have delivered the message. In order to perform a reliable broadcast of a message m, a process first delivers m, intializes its got variable to itself and then starts to execute the diffuse task in the background. This task at process p periodically checks if for some neighbor not in the got set for the message, the heartbeat of that neighbour has increased, and if so p sends the message marked with the got set and its own ID to all neighbours whose heartbeat increased. Diffuse is done when all neighbours of p are in the got set.

If a process receives one of these messages with a got set it checks if it has already delivered m. If it did not it delivers the message and starts the diffuse task and sets the got set to itself. Then, regardless of whether or not it had delivered the message previously, p adds the received got set to its own and appends its path to the received one and sends the message with the new got set and path to all its neighbours that appear at most once in the path.

For proofs of correctness:

Uniform Integrity, Validity, and Partition Integrity are all pretty obvious from the code. (Uniform integrity is satisfied because processes check if they have delivered a message previously and do not deliver one again if they have. Also, the only way for a message to be delivered is if it was sent. Validity holds because a process delivers a message first thing. Partition integrity holds because of properties of links in this network setting.)

Agreement:

For agreement, note that for a correct process p in the partition, there is a fair path from p to every other process q in the same partition. Now all we need to show is that if a link from process p to q in the same partition is fair, then if p delivers a message m then q eventually also delivers it. Repeated applications of this lemma give us agreement.

We'll do this by contradiction. Suppose p delivers m but q never does. Since q does not deliver m, it must be the case that q never belonged to the got set of m at p. By the HB Accuracy property we know that the heartbeat sequence of q at p is unbounded, so p must resend the message to q an infinite number of times. Since the link is fair, q must eventually receive the message and then deliver it, a contradiction.

The final portion of correctness for this algorithm lies in showing that the algorithm is quiescent. We need to show that a single invocation of broadcast only sends a finite number of messages.

Quiescence helper:

We first show that if we assume that q is a neighbour of p in the same partition, that if p forks a diffuse(m) task then eventually q will be in the got set of m at p.

Since p and q are in the same partition, we know there is a fair path from p to q and from q to p.

For each step of this path short of returning to p, we need to show that there is a got set containing all processes prior to the step and such that the process at the step sends a message with this got set to the next process in the path an infinite number of times. This will produce a contradiction since links are fair and we can conclude that p eventually adds q to its got set when it unions the received set in line 23.

This is an inductive proof. For the base case we know that, since q never appears in p's got set for m despite being a neighbour and having a nondecreasing and unbounded heartbeat, p will repeatedly send out messages to neighbours with increasing heartbeats, including the next process on the fair link path to q. So p must send to this next process a message with its id as the path an infinite number of times. Since there are only a finite number of different messages of this form that could be sent, there must be some set of processes sent as the got set an infinite number of times. This set will necessarily contain p.

For the inductive portion of the proof, say that for some step j of the fair path from p to q to p again, that p_j sends the message with a set g_j containing all processes so far on the path and a path P_j to the process at step j+1 an infinite number of times. By fairness of links, we know that the process at step j+1 receives this message an infinite number of times. Since the process at step j+2 is a neighbour and can only appear once in P_{j+1} , then each time the process at j+1 receives the message it sends a message to the process at step j+2 with a got set containing itself and g_j. Since a message is sent to this next process an infinite number of times, we're done.

This immediately gives us that if a correct process executes a diffuse of a message, it eventually stops sending messages for that task since either the above case holds or the neighbour is not in the partition and the heartbeat failure detector eventually stops increasing and the correct process will not send any more messages in the task.

Quiescence:

This finally leads us to quiescence. Assume for contradiction that every process does not stop sending messages for m. Then there has to be some process p such that p never stops sending messages for m. Since there are only a finite set of values for the path and got set to range over then there must be some fixed path that p sends an infinite number of times. Pick the shortest path such that some process sends the path an infinite number of times attached to one m. We have already shown that there is some time when that process stops sending messages for the diffuse task. As a result, this process must be sending an infinite number of messages in the only other place it sends messages, namely, line 26. Each of these sends must be preceded by a receipt of a message with m and a path one shorter than the one p is sending out with messages. For this to have happened an infinite number of times, some process had to have sent it an infinite number of times. But this path is shorter than the one process p is sending out, a contradiction.

Nice abstraction:

This implementation of reliable broadcast actually has a pretty nifty abstraction... notice that in the algorithm processes do not need to know the identity of all processes in the system, only its neighbours. Also, processes only need to maintain information about heartbeats of its neighbours. I believe this actually means that for the purpose of this algorithm the implementation of the heartbeat failure detector module at each process only has to maintain information about its neighbours.

Quasi-reliable send and receive

Section 4.3

Finally we are left with the last two communication primitives to define: qr-send and qr-receive. These quasi-reliable sends and receives, remember, define a send and receive from one process to another. To implement these operations according to the properties of quasi no loss, uniform integrity, and partition integrity discussed before, we can do the following:

To qr-send a message m from p to q we broadcast the message (m, p, q, k) using reliable broadcast. The number k is a sequence number for the message, a number that process p has not used before. Note that this sequence number is just to make the message unique for reliable broadcast... the message itself may have been qr-sent before to the same destination. Upon delivery of this four-place message, a process r qr-receives m if its id is the same as that of q in the message and simply discards it otherwise. Since reliable broadcast is quiescent, this plainly is as well. All other properties are obvious from the properties of reliable broadcast (it's just reliable broadcast except that we only qr-receive if it was for us).

4 Scalability and service

Applications for real large scale distributed systems has forced an interest in failure detector algorithms that minimize network load and process computation while still providing certain quality of service guarantees with respect to completeness, accuracy and speed. The required level of quality of service should vary according to application specified constraints.

The scalable and efficient failure detectors paper by Gupta, Chandra, and Goldszmidt focuses on crash recovery systems that suffer probabilistic link failures and process failures. Different papers adopt a different focus when it comes to this discussion. For example, this paper characterizes process failures probabilistically but other papers, such as one by Chen, Aguilera, and Toueg, claim that occurrence of process failures should be considered as a purely nondeterministic thing. That other paper also concerns itself with more than just probability of link failure, but also message delay, claiming that this can be an important consideration since at various times of day a server might experience much larger message delivery delays. However, this paper we are considering does not concern itself with message delays, saying that in most systems several seconds is all that might be expected at worst as far as such a delay goes.

4.1 The Model

Section 3

We're going to consider a large group of n members, fixed beforehand. Each member has a unique identifier and maintains a list containing the identifiers of all other group members. There are extensions of the study we are about to consider to the situation where membership is changing and members do not have complete lists of other group members, but we won't consider that here. Clocks are not synchronized between processes, but the drift rate of the clocks remains constant. Maximal message sizes are constant.

Processes have an incarnation number, stored as an integer in nonvolatile storage, that is incremented every time the process recovers. After a process crashes, it is detected at exactly the first time that a nonfaulty member detects a failure of the process in an incarnation at least the same as the incarnation when it crashed or a recovery of the process with an incarnation strictly larger than that of its failure.

Member crash probability, p_f, is the probability that a random group member is faulty at a random time. This is independent across members.

Message loss happens with probability p_{ml}. This distribution is also independent across messages. The paper points out that this might be suspect, but that if failure detection protocol periods are much larger than message propagation time, then messages exchanged have large temporal separation, reducing message loss correlation. Randomized message destinations, which will be used in a little bit, also weaken the correlation.

4.2 Scalability and Efficiency

Section 4

Completeness properties required by most distributed applications have led to the use of failure detectors that guarantee eventual strong completeness but only guarantee strong accuracy with some probability less than 1.

Requirements imposed by an application can be specified as follows:

1. Completeness: Crash failures of processes should eventually be detected by all non-faulty processes in the group.

2. Efficiency, which is twofold:

a. Speed: In many distributed applications it is not necessary that a member failure be known to all group members, but rather to at least one. In many systems, just a few computers or a central system aggregate failure detection information across a system. So it makes sense to consider efficient detection as depending on when one non-faulty member identifies a process as failed. This leads us to defining the speed requirement as follows: A member failure is detected by some non-faulty group member within T time units after its occurrence. We'll assume that T is much larger than the worst case message roundtrip time. This value T is specified as a quality of service parameter.

b. Accuracy: At any time instant, the probability for every nonfaulty process that no other non-faulty process mistakenly identifies it as faulty within the next T time units is at least (1-PM(T)). This value PM(T) is specified as a quality of service parameter.

These quantifications are similar to the primary metrics identified in the quality of service paper of Chen, Toueg, and Aguilera. That paper considered detection time, related to speed, which gives an expected length of time from a process crash to its detection and two accuracy metrics, mistake recurrence time and mistake duration. Mistake recurrence time is the time between two consecutive mistakes. Mistake duration is the time it takes a failure detector to correct a mistake. The property of accuracy described in the paper we are studying right now is very similar to the derived property of query accuracy probability in Chen, Toueg, and Aguilera's paper. In that paper, query accuracy probability is just one of several derived metrics for accuracy, derivable from the more basic mistake recurrence time and mistake duration.

In addition to satisfying requirements of applications, we are concerned with providing failure detectors that do not impose large network load. To measure scalability we consider the worst case network load imposed by an algorithm, denoted L.

Worst case network load: This is the largest number of messages transmitted by any run of the protocol within any time interval of length T, divided by T.

We also want failure detectors to impose uniform send/receive loads at members.

This leads us to a definition of a property that we want satisfied, scale:

Scale: the worst case network load L imposed is close to the optimal, and expected load per member is uniform.

4.3 Network Load

This is all fine and good, but what is the optimal worst case network load, which we'll call L^*, given the need for completeness and the parameters for speed, T, and accuracy, PM(T)? As long as we assume that the number of processes is large and PM(T) is much smaller than the probability of message loss, p_{ml}, then we have the following theorem from p. 173:

Theorem 1: Any distributed failure detector algorithm in a system with message loss probability p_{ml} that deterministically satisfies completeness, speed, and accuracy for the given values of T and PM(T) imposes a minimal worst case network load of:

$L^* = \frac{nlog(PM(T))}{Tlog(p_{ml})}$

Proof: To prove this we need to show that each nonfaulty group member in a time interval of length T could transmit $\frac{log(PM(T))}{log(p_{ml})}$ messages.

Consider a group member i at a random time t. Let it not be detected as failed yet by any other group member and let it be nonfaulty for the next T time units. Let m be the maximum number of messages sent by i for these T time units in any possible run of the failure detector protocol starting from t.

The probability that all messages sent by i in this time interval are lost is p_{ml}^m, since there are m messages, each lost with probability p_{ml}. If this happens, then it is indistinguishable to other group members whether or not i has failed. By the speed requirement, it will be detected as failed by some member in this interval. So the probability that an as-yet nonfaulty member is detected as failed in T time units is at least p_{ml}^m. By the accuracy requirement, $p_{ml}^m \leq PM(t)$. This implies that $m \geq \frac{log(PM(T))}{log(p_{ml})}$.

We can actually quickly sketch out an algorithm that realizes this theorem:

A highly available server or set of servers are designated as leader. All other members send $\frac{log(PM(T))}{log(p_{ml})}$ messages to the server every T time units. The server labels a process as failed if it does not receive any messages from a process for T time units. It's plain to see that this satisfies the requirements.

At this point we'll introduce a new term:

Sub-optimality factor: this is defined as L/L^*. We want this factor to be independent of group size.

An interesting point to discuss now is why so-called traditional heartbeat schemes are not optimal. These algorithms have every group member periodically send out a heartbeat message with an incremented counter to all other group members. A member is declared as failed by a nonfaulty member if the nonfaulty member does not receive these heartbeat messages from it for some number of consecutive heartbeat periods. In the simplest implementation of this scheme, members transmit alive messages to everybody every T time units. This gives us n messages a time unit and a worst case network load of n^2. The suboptimality factor varies as n. Here, the algorithm does not scale. These heartbeating schemes attempt to communicate with all members. With a centralized set of servers, we can avoid a great deal of this traffic. Other variants of heartbeat schemes can be configured to have optimal load, but suffer other sorts of performance problems, such as undue load at certain processes.

4.4 A Configurable Randomized FD Algorithm

Section 5

Now we will go on to describe a randomized distributed failure detector protocol that imposes equal expected load per member, rather than selecting a central group of servers. We relax the speed condition to detect failures in expected time rather than exact time as before. This algorithm satsifies completeness with probability 1 and has an accuracy probability of 1-PM(T). The sub-optimality factor of this algorithm is independent of group size.

The algorithm has two parameters: a protocol period, T', and an integer k, for the size of failure detection subgroups. These can be configured from the application provided factors T and PM(T), and the network parameters p_f and p_{ml}. We'll discuss that in a minute. T' and k are known by all members.

[Algorithm on Figure 1, p. 175.]

In the protocol, a member M_i randomly selects another member M_j to ping. If it does not receive a reply in a period as long as the worst case round trip message time, then it selects a random k members to outsource ping-req messages to. Each of these nonfaulty processes that receive such a request ping M_j and forward any received ack back to M_i. If M_i waits until the end of the period and does not receive any acks, M_j is declared as failed.

Note that asking k other members to ping M_j has better behavior than having M_i ping M_j k times (i.e. if the link between M_i and M_j fails), though this additional benefit is not explicitly represented in the authors' model.

4.4.1 Analysis

Now we analyze the algorithm.

We want to calculate the values of T' and k as functions of T, PM(T), p_f, and p_{ml}.

For convenience, we define $q_f = 1 - p_f$ and $q_{ml} = 1 - p_{ml}$.

For any group member M_j, the probabilty that anybody chooses to ping it directly in a time interval T' is:

$= 1 - Pr[\text{nobody pings it}]$

$= 1 - Pr[\text{one random process doesn't ping it}]^{n-1}$

$= 1 - (1 - Pr[\text{one random process does ping it}])^{n-1}$

$= 1 - (1 - (Pr[\text{random process doesn't fail}]/\text{number of processes it has to choose from}))^{n-1}$

$= 1 - (1 - q_f/n)^{n-1}$

Since we are assuming n much larger than 1, this reduces to $1 - e^{-q_f}$.

So the expected time between a failure of a member and its detection by some nonfaulty member is:

$E[T] = T'/(1 - e^{-q_f}) = \frac{T'e^{q_f}}{e^{q_f}-1}$

So we have a way to calculate T' from T and p_f. If we call $\frac{e^{q_f}}{e^{q_f}-1}$ $C(p_f)$ then $T' = E[T]/C(p_f)$.

To calculate PM(T), we just note that this is the same as the probability that there is at least one M_l s.t. nonfaulty M_j is detected as faulty due to unlucky message loss (none of the direct or indirect pings from M_l have corresponding received acks).

A random member M_l is nonfaulty with probability q_f. The probability of such a process choosing to ping M_j in a time interval of length T is $C(p_f)/n$ (there are n processes it could have chosen to ping and $C(p_f)$ time intervals of T' in T). The probability that M_l receives no acks is:

$(1 - q_{ml}^2)((1 - q_{ml}^4 q_f)^k)$

This is the chance that a direct ping request fails by message loss either way multiplied by the chance that, for each of the random k outsource processes, the links fail on any of the four legs of the message journey or the outsource process fails.

This gives us that

$PM(T) = 1 - [1 - \frac{q_f}{n}C(p_f)(1 - q_{ml}^2)(1 - q_{ml}^4 q_f)^k]^{n-1}$

Since n is much larger than 1 we can reduce this to:

$1 - e^{-q_f C(p_f)(1-q_{ml}^2)(1-q_{ml}^4 q_f)^k}$

Given that PM(T) is much smaller than 1 we can reduce further to:

$q_f C(p_f)(1 - q_{ml}^2)(1 - q_{ml}^4 q_f)^k$

This finally gives us a way to calculate the value of k:

$$k = \frac{log(\frac{PM(T)}{q_f(1-q_{ml}^2)\frac{e^{q_f}}{e^{q_f}-1}})}{log(1-q_{ml}^4 q_f)}$$

Hence, we can conclude that the failure detector protocol can be configured to satisfy speed and accuracy requirements. Also, given a member that has failed, every other nonfaulty member will eventually choose to ping it in some protocol period (probability 1) and hence it will be discovered as failed. The processes all have uniform send/receive load. The only result remaining to be shown is that of scale. We need to upperbound worst case and expected network load.

We'll do worst case first. The worst case happens when a member pings on line 1 and 6, which is 2 messages, and then taps k other processes to ping (4 messages each). This gives us:

$L = n[2 + 4k]/T'$

Substituting out k and T' using the equations we just got done constructing, and using Theorem 1, we get a suboptimality factor that is independent of group size. Without going over the figures since it is not really enlightening, the important thing to note is that the factor of n in L is cancelled out when it is divided out by the n of L^*. Since k and T' are not in terms of n, we know that this final result will not be either.

To calculate the average network load imposed by the algorithm, we note that the only differences between worst case and average are as follows:

1) rather than n processes executing, we have nq_f, since some may fail

2) rather than always executing steps 2 - 5, we will only have to execute them when no ack is received from the target of the ping. This only happens with probability $1 - q_{ml}^2 q_f$, or the probability that the ping is lost on either leg of its journey or that the pinged process has actually failed. This gives us that the average expected network load is:

$E(L) = nq_f[2 + (1 - q_{ml}^2 q_f)4k]/T'$

Again, we could work out the suboptimality factor, but the result is just a jumble of math and the important part is that again the group size, n, is factored out, leaving us with expected load that is independent of group size. Hence, this algorithm achieves the requirement of scale.

One aside that may be of interest is that in the QoS paper of Chen, Toueg, and Aguilera, the authors discuss the fact that the probability of message loss might vary over the course of execution and suggest the use of very simple heartbeat schemes that can be used to easily estimate the message loss probability. These schemes could be executed periodically to adapt to changing network conditions, though some work would be needed to make it fit into this particular scheme.

5 Recap

Just to recap quickly, we've gone over some of the common terminology from the failure detector community and seen how some of these failure detectors can perform work in less than perfect network conditions. We've also been introduced to the heartbeat failure detector, a simple idea that extends the work that it is possible to complete in asynchronous systems. Finally, we examined some basic notions of scale and configurability of failure detectors that provide a foundation for real work in large systems with processes that have differing requirements for a failure detector service.