Analyse van artikel

“It is clear that the Visited or Token messages can be sent at most once over each link in either direction.” – This can be seen from the process description (step 1) by the decreasing two sets: the candidate set and the information set. Since two processes have their own local state, and is initialized with full neighbor knowledge.

“This implies that no more than four messages are sent over any link in both directions.” True.

“To prove that at most three messages are sent, it is enough to show that: if two Visited messages are sent, then at most one Token message is sent (since at most two messages of each type may be sent, this also implies that if two Token messages are sent, at most one Visited is sent).”

H: Two Visited messages are sent, i.e. (P sends *info* to Q) and (Q sends *info* to P).
C: At most one Token message is sent, i.e. (P sends *token* to Q) or (Q sends *token* to P).

“Consider the case that two Visited messages are sent over link (P,Q)”. H.

“Also assume that a Token message was sent first, from P to Q (which implies that P is not the son of Q).”

H: A Token message is sent from P to Q. Son is what we here call “intended candidate”, or “opvolger”. If P sends to Q a token as its intended candidate, P indeed never sends an *info* message to Q. This is not the case, since we assumed two *info* messages, hence P can never have Q as intended candidate. We can furthermore assume that P sends a Token message to Q, but that can only happen in the case described below: the *info* from Q to P is not received when P sends that token to Q.

“In this case, the Token is sent by P before receiving the Visited message of Q, otherwise no Token would be sent to Q”. Clear: this is precisely the same reason as above.

“Since Q sends a Visited message over PQ, this cannot be the first Token received by Q.” Correct: Q needs to have a different intended candidate before it can send an *info* message to P. However, the *info* message from P must also not be received by Q, otherwise it would not send an *info* message to P.

“In such a case, when the Token sent by P is received at Q, channel QP is marked (by Q) as visited or unvisited, so no response is sent.” H: Q receives the token by P. If the channel is already visited (the scenario we have below) then Q indeed does not send any response: it already has forwarded the token to P before, and before that it has sent an *info* message. Any received token will then be ignored. If the channel is still unvisited, and Q receives a token by P, there are two cases: (1) Q has not yet received a token before, ruled out since Q must also send an *info* message to P, (2) P is still a potential candidate for Q or P is the intended candidate of Q. If P is still a potential candidate for Q, P may also be the intended candidate for Q. Four cases: (2.1) P is the intended candidate for Q, then the reception of the token by P is interpreted as a normal token return (see step 9), (2.2) P is a potential candidate for Q, and the reception of the token is from a non-intended channel and is interpreted as an information message (see step 9), (2.3) Q has already replied back to its parent, and the token is ignored. In all three cases, there will be no response, (2.4) P is no longer a potential candidate for Q, then either we received the information message or the token was already forwarded to P and the information message resulted in selecting a next intended candidate.

“Moreover, after the Token reception QP is marked as visited, no future Token message may be sent by Q to P.” That is (literally) correct: P is always removed from the potential candidates from Q. However, what is missing from this statement is the past: Q may have already sent a Token message to P before it received a Token message from P.

“We have proved…” is not correct. Let’s consider the proved case:

H: Two Visited messages are sent
H: A Token message is sent from P to Q
H: Q receives the token by P
C: No future Token message may be sent by Q to P

This is correct (and stated by the article). However, what is missing to prove the actual statement “at most three messages are sent over each link” is the past:

H: Two Visited messages are sent
H: A Token message is sent from P to Q
H: Q receives the token by P
C: No past Token message was sent by Q to P

It will be hard to prove this statement (see below for a counter-example).

Suppose two visited messages are sent, and P has sent a Token message to Q, and Q has received the token by P. As was identified above we have three cases: (2.1) P is the intended candidate for Q, then the reception of the token by P is interpreted as a normal token return (see step 9); here Q has sent the token to P, from which the conclusion is not derivable (2.2) P is a potential candidate for Q, and the reception of the token is from a non-intended channel and is interpreted as an information message. Since P is still a potential candidate removed only by this information message, no past Token was sent by Q to P. (2.3) Q has already replied back (i.e. it’s potential candidate set is empty) to its parent, then in only when the *info* message from P was received before P became the intended candidate for Q, was there no past Token message from Q to P. However, if the *info* message arrived after P became the intended candidate for Q, a token was sent and a new candidate will be picked. (2.4) P is no longer a potential candidate for Q, same reason as for (2.3).

Analyse van tegenvoorbeeld (Exercise 4.2 uit boek)

“Give an example of a computation of Cidon’s algorithm in which two information messages and two tokens are communicated through the same channel in the network.”

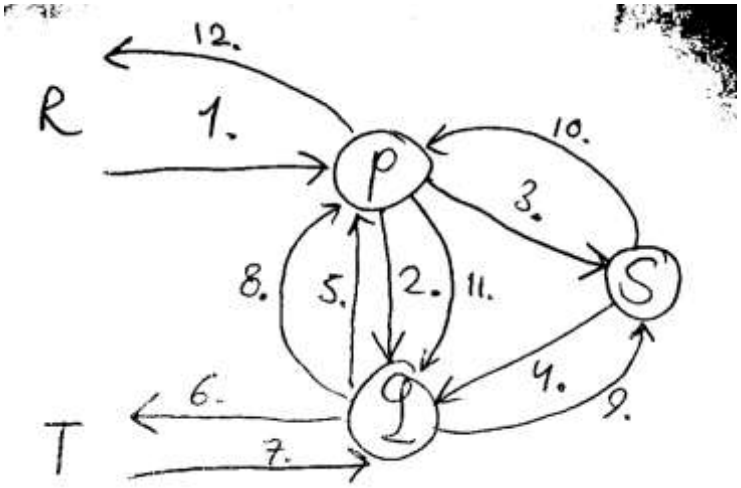
Het voorbeeld waar ik aan denk is het volgende:

Een netwerk, waarbij we twee aan elkaar verbonden nodes (P en Q) met daartussen een channel bekijken, zodat channel PQ vier berichten ontvangt:

1. P ontvangt van een andere node, zeg R, de token. Dit is de eerste keer dat P een token ontvangt, en dus wordt de parent van P gezet op R en wordt van de overgebleven buren, de potentiële kandidaten, een opvolger gekozen: dit wordt er een anders dan Q, dus zeg S.
2. Omdat P de opvolger S heeft gekozen, worden de andere buren (behalve dus parent en opvolger) met een *info* message ingelicht: dus P stuurt *info* naar Q. Er wordt niet gewacht op een acknowledgement.
3. P stuurt de token door naar S.

4. S stuurt de token door naar Q. Zo ontvangt Q voor de eerste keer de token, dus de parent van Q wordt gezet op S. Van alle overgebleven buren van Q kiest Q een opvolger, anders dan P, dus zeg T.
5. Het info bericht van P is nu nog niet ontvangen. Hierdoor dient Q voor alle buren behalve opvolger en parent een *info* bericht te sturen: dus Q stuurt *info* naar Q. Er zijn nu dus twee info berichten onderweg op het channel PQ.
6. Q stuurt de token door naar T.
7. T stuurt de token terug naar Q.
8. Q ontvangt de token (dit maal niet als eerste), en kiest een nieuwe opvolger: dat wordt P. De token wordt naar P opgestuurd. Momenteel onderweg op channel PQ: *token* van Q naar P, *info* van Q naar P en *info* van P naar Q.
9. Q ontvangt *info* van P. Hierdoor weet Q dat de vorig verzonden token wordt genegeerd, en een nieuwe token gaat terug naar S.
10. S stuurt de token terug naar P.
11. Momenteel onderweg op channel PQ: *token* van Q naar P en *info* van Q naar P. Hierdoor weet P dus nog niet dat Q de token heeft gezien. P stuurt dus de token naar de enige overgebleven kandidaat: Q.
12. P ontvangt *info* van Q en daardoor weet P dat de token wordt genegeerd. Alternatief: P ontvangt *token* van Q en deze wordt als antwoord op de token van 11 gezien. In beide gevallen: P stuurt token terug naar R.

PQ heeft nu 4 berichten.



Notities tijdens coderen van Cidon

An adaption of Awerbuch's algorithm, by no longer waiting for acknowledgements of information messages. A process forwards the token without delay, that is, without waiting for the acknowledgement of information messages. The process to which the token was forwarded last is recorded, and whenever a token is received that is not equal to the last forwarded, the token is purged (ignored), and the edge is removed as potential successor candidate (no longer a tree edge, i.e. a frond edge). Suppose that some process, q, has forwarded a token and later receives an information message from the process, p, it forwarded the token to: in that case, the edge pq is marked as a frond edge and a new candidate (if any) is selected and a new token is forwarded from q.

If we analyse the description of this algorithm, and compare the state space and description with that of Awerbuch's, we observe that:

- There is no need for keeping track of the processes to wait for an acknowledgement,
- There is also no need for sending back acknowledgement messages,
- The potential candidate now is remembered even after sending the token,
- Receiving an information message not only clears the channel from the successor set, but may also resend a token to another candidate (or back to its parent) if the message came from the remembered candidate.

Hence the process state space is the tuple: (Bool, ReceivedUnseen<Channel> | ReceivedSeen<Channel> | Replied<Channel> | Undefined | InitiatorUnseen | InitiatorSeen, Set<Channel>, Maybe<Channel>, Set<Channel>, Set<Channel>, Maybe<Channel>).

At this point the description in this document of Awerbuch and the actual implementation has diverged (see the remarks after experimentation above for all changes). We should investigate a structural solution for writing down process descriptions, as the coding should be a reliable source for reading and understanding the implementation. Additionally, it is expected that with more complicated algorithms, we need more structure to even cope with the coding process.

We now write down an adapted version, based on the implementation of Awerbuch:

State space:

- Does the process have the token? (Bool)
- What state is the process in? (RRUI, note that it now seems cleaner to use an Info flag, mimicking the Appendix of the book)
- A channel intended to forward the token to after the information messages have been sent (Maybe Channel, this differs from Awerbuch as it is not cleared after sending the token)
- A set of neighbor potential token forwarding channels (Set of Channel, all except parent and chosen channel and received information messages)
- A set of information messages still to be sent (Set of Channel, this seems a common idiom)

Is the last channel from which a token was received still necessary? In the DFS algorithm, it was stored to indicate a direct return of the token. In Awerbuch this field is also not strictly necessary, since the algorithm already is a depth-first search without this state. Let us reason why this is the case: if a token was already received, in the case of Awerbuch an information message is sent to that process; thus a token ping-pong, where it is sent and immediately returned if it traveled over a frond edge, never occurs. Since we never directly send back a token, unless there are no further children (either because all edges are frond edges, or because there are no other edges), the state is not necessary. Compare this to Cidon's: we explicitly need to keep track of the intended channel to ensure that receiving an information message after sending the token results in passing the token on to a new candidate. This field is the same as the last channel field of the DFS algorithm. We thus drop the redundant last channel field.

Also, since a process only sends information messages once, it seems unnecessary to have two sets for the channels where to send the information messages to. Also note that while the set of

information message is not empty, we may consider the process as not having seen the token for the first time. Thus the Boolean flag for seen the token is redundant too.

The initial state space for an initiator becomes:

- True, it has the token,
- Initiator, that remains fixed for the whole algorithm,
- Nothing, no intention yet,
- A set of all neighbor channels, potential forwarders,
- A set of all neighbor channels, for information messages.

A noninitiators has the initial state of:

- False, not having the token,
- Undefined, not yet a parent,
- Nothing, no intention yet,
- A set of all neighbor channels, potential forwarders,
- A set of all neighbor channels, for information messages.

Process description:

1. (True, Received ch|Initiator, Nothing, p, i) and (size p) > 0, then we choose an intended channel: in = next p. We assume that both p and i do not contain the parent channel (it must be removed upon receiving a token for the first time). The next state becomes (True, Received ch|Initiator, Just in, remove p in, remove i in) by an internal event. It is an invariant that i is a superset of p before a token was seen (signaled by the Nothing), hence checking size of p is sufficient.

2. (True, Received ch|Initiator, Just in, p, i) and (size i) > 0, we send an information message to im = next i, and the next state becomes (True, Received ch|Initiator, Just in, p, remove i im).

3. (True, Received ch|Initiator, Just in, p, i) and (size i) = 0, we have sent all information messages and thus forward the token to in, with the next state of (False, Received ch|Initiator, Just in, p, {}).

4. (True, Received ch, Nothing, p, i) and (size p) = 0, (size i) = 0, we have sent all all information messages and we have no successors left, thus we send the token to ch, we have no longer an intention of sending any token that may fail, and the next state becomes (False, Replied ch, Nothing, {}, {}). A special case is receiving a token on a dead-end, namely by directly returning it.

5. (True, Initiator, Nothing, p, i) and (size p) = 0, (size i) = 0, we have sent all information messages and have no successors left. Decide.

6. (False, Replied ch, Nothing, p, i) and (size p) = 0, (size i) = 0, we have sent all information messages and have no successors left. Terminate.

7. (False, Undefined, Nothing, p, i), and if we receive a token from ch the next state becomes (True, Received ch, Nothing, (remove p ch), (remove i ch).

8. (False, Received pa|Initiator, Just in, p, i), and if we receive an information message from ch, then if ch = in, we have sent a token over a frond edge and choose a new successor by having the state (True, Received pa|Initiator, Nothing, p, i). Otherwise, we interpret it as an information message an choose the state (False, Received pa|Initiator, Just in, (remove p ch), i).

9. (False, Received pa|Initiator, Just in, p, i), and if we receive a token from ch, and only if ch == in, we handle the token as a normal reply with state (True, Received pa|Initiator, Nothing, p, i). Otherwise, we interpret the token as an information message with state (False, Received pa|Initiator, Just in, (remove p ch), i).

10. (False, Received pa|Initiator, Nothing, p, i), and if we receive a token, we handle the token as normally with state (True, Received pa|Initiator, Nothing, p, i).

11. (False, Undefined|Received pa|Initiator, Nothing, p, i) and we receive an information message from ch, then we have state (False, Undefined|Received pa|Initiator, Nothing, (remove p ch), i).

Remarks after experimentation

We noted that step 8 and 9 are the same.

There is an open question regarding the following case:

Suppose some process sends an information message to a process. If the information message arrives first, then that process removes the sender as a potential neighbor. If it then receives the token, the information message is discounted and the token will eventually be returned. However, if the token arrives first then the process may terminate before it has received the information message. Hence we arrive at the following problematic situation: may a process terminate if it has not yet received all its incoming messages?

This problem may become important once algorithms can be composed to form larger algorithms. Do we assume that all messages sent by a previous algorithm that is sequentially composed to the next algorithm arrive, or do we assume that the messages from different compositions are disjoint and hence the old message space can be safely discarded?

Now suppose that we require that an algorithm has received all messages before terminating (we are not talking about the deciding process, but all processes). Then, in the case of Cidon's algorithm, a local process cannot decide for itself when to terminate. The previous case exemplifies this:

1. A process never knows whether it was chosen by its predecessor before sending information messages or was a successor after all information messages are sent. We can see this behavior, e.g. in example 4.3. Here process r receives the token first from q, and thus sends an information message to all neighboring processes except its parent (q) and its decision where to send the token next (p). Thus, r has sent an information message to s and to t.

2. Then, eventually, r receives the information message from p, and thus recreates the token and send it to s. Now we potentially have two messages in the channel sr: an information message and a token.

3. The same applies to t, after s has returned the token to r, in that the channel rt may contain two messages: an information message and a token.

4. Note that we cannot decide, at process t, how many information messages it can expect to receive. We already saw the possibility that t could receive a single information message from r. Now

suppose that r could have decided to send the token directly to t (and thus only send information messages to p and s). This choice is non-deterministic and made externally.

We have two options:

1. we can always send an information message (and thus change Awerbuch's algorithm). The algorithm already copes with the fact that an information message may precede a token, hence requires no adaption on the receiving side. The algorithm may (locally) terminate only if it has received all information messages from neighbors.

2. We can introduce two sequential composition operators: the first, let's say union, requires that the two algorithms are disjoint and that their union does not interfere in the message space (hence requires proof for every local termination that all messages are received). The second, say disjoint union, lifts this restriction and tags each message space to correctly disambiguate late message and drops them upon reception after local termination of the first algorithm.

Since we require the implementation of the original algorithm, it seems that 2 is also necessary. Still, it seems useful to explore this alternative to Cidon's algorithm in the implementation.

ERRATUM Page 22 of revised first edition. "In Cidon's algorithm, frond edges may carry two information messages and two acknowledgements (see exercise 4.2)". Cidon's algorithm does not have acknowledgements. Instead, it should read "may carry two information messages and two tokens".