CS142 Homework Set #7 Solutions

Timur Kuzhagaliyev

November 22, 2017

Problem 1

Intuitively, the algorithm should work because the logic described in the question is equivalent to running two instances of the 1-bit algorithm simultaneously and in perfect sync, so the messages could be superimposed.

The variables for officers remain the same as in proof used in class, except now $commit[u, r]$, $confirm[u, x, r]$ and $confirmationCount[u, r]$ operate on vectors of size 2, updating relevant vectors component-wise. Initially, the values for each commander are $(false, false)$, $(false, false)$ and $(0, 0)$ respectively, and general makes the decision to commit following the logic described in class, but now applied to each bit separately.

The officers now can also send out multiple commit messages, one for committing each bit while the other bit is uncommitted. Alternatively, if the officer commits both bits at the same time, a single commit message is sent. If an officer receives multiple commit messages from the some other officer, it takes the a logical OR of all messages.

I will now show that all of the lemmas and theorems defined for the 1-bit version of the algorithm hold for the 2 bit version.

1. Lemma 1: $confirm[u, v, r] = commit[v, r - 1]$.

   Proof: If a non-faulty general v commits one (or both) of its bits in round $r - 1$, it will send out a $commitMessage[v]$ to all other officers, containing the values for each bit. This means that all non-faulty generals u would have received $commitMessage[v]$ by round r, and hence $confirm[u, v, r]$ holds (for the relevant component).

2. Lemma 2: $confirm[u, x, r - 1] \Rightarrow confirm[v, x, r]$ where u, v are non-faulty officers.

   Proof: When u received a 2-component $commitMessage[x]$ in $r - 1$, it has relayed it to all officers including v. In round r, v has received $commitMessage[x]$ so $confirm[v, x, r]$ holds (for the relevant components).

3. Theorem 1: If a non-faulty officer commits one (or both) of the bits in round r, then all non-faulty officers commit the same bits by round r + 1, where the comparison is defined component-wise

   Proof: We know that $commit[u, r] = commit[u, r-1] \vee ((confirmationCount[u, r] \geq r) \wedge confirm[u, g, r])$. Suppose u commits one (or both) bits for the first time in r, so the first part of the condition must be false. In this case, for u to commit some bits it must have received at least r commit messages from other officers and a commit message from the general for the same bits. Now, u will relay all of the commit messages it has (including the general's and its own commit) to other generals.

This means that by round $r+1$, all non-faulty officers v would have received $r+1$ commit messages and a commit message from the general for some bits, so $confirmationCount[v,r] \geq r) \wedge confirm[v,g,r]$ would hold and they would commit the same bits.

4. Theorem 2: If no non-faulty officers commit one (or both) of their bits in the first t rounds, then no non-faulty officer will commit the same bits in later rounds, where t is the maximum number of faulty generals.

   Proof: We know that $commit[u,r] = commit[u,r-1] \vee ((confirmationCount[u,r] \geq r) \wedge confirm[u,g,r])$, similarly: $commit[u,t+1] = commit[u,t] \vee ((confirmationCount[u,t+1] \geq t+1) \wedge confirm[u,g,t+1])$, for each component of the vector separately. By assumption, $commit[u,t]$ does not hold for the relevant bits, for all non-faulty officers u. This means that each non-faulty agent can have a maximum of t commit messages for the relevant bit (from the faulty officers), and at round $t+1$ the condition $confirmationCount[u,t+1] \geq t+1$ will not hold. Hence $commit[u,t+1]$ will be $false$, for round $t+1$ and all subsequent rounds (if any).

The proofs above demonstrate that the algorithm will indeed work, much like the algorithm discussed in class.

Problem 2

The algorithm would not work if forgery is allowed. The original version, which relied on the fact that forgery was disallowed, made heavy use of the fact that the message from the general could not be forged (i.e. $confirm[u,g,r]$ would always be what the general originally sent, whether the general is faulty or non-faulty).

We will exploit this fact to show how the safety property can be violated. Consider the case where the general is non-faulty and it sends out a command to retreat. Our safety property dictates that all non-faulty officers must obey this command.

Now, the 2 faulty commanders can work together - they can both modify the general's message to make it look like the original command was attack, and then send out their commit messages and the forged general message to some non-faulty commanders during round 1.

In round 2, these non-faulty commanders would have received at least 2 commit messages (from the faulty commanders) and the (forged) attack message from the general, hence they will commit. This violates the safety property as the non-faulty commanders disobeyed the order of the non-faulty general, and they cannot change their decision.

Therefore the algorithm is not correct in the case where forger is allowed.

Problem 3

a) Types of messages used in the Paxos algorithm:

1. **prepare(n)**. This message type has a single argument n which is a value in a total order. In this code n is a pair (t, id), where t is a non-negative integer that can only increase and id is the static ID for a particular agent, and the comparisons in total order are made lexicographically. To simplify the descriptions in the future I'll refer to pairs (t, id) in the total order as "timestamps".

   Message type **prepare(n)** is sent by a proposer and is received by an acceptor.

2. **granted(n, latest_accepted, latest_accepted_n)**. This message type has three arguments: n and *latest_accepted* are timestamps, *latest_accepted* is some value representing the proposal that was last accepted by the acceptor. If *latest_accepted* is not $None$, then *latest_accepted_n* contains the timestamp for which the proposal was accepted. n contains the value of the corresponding (successful) **prepare(n)** message.

   Message type **granted(n, latest_accepted, latest_accepted_n)** is sent by an acceptor and is received by a proposer.

3. **nack_prepare(n)**. This message contains a single argument n. It is used to notify proposers that their message **prepare(n)** was not successful.

   Message type **nack_prepare(n)** is sent by an acceptor and is received by a proposer.

4. **accept(n, proposal)**. This message contains two arguments: n is the timestamp that corresponds to the last **prepare(n)** message the proposer sent (and subsequently the returned **granted(n, ...)** message), *proposal* is some value that the proposer wishes to propose (subject to constraints described below).

   Message type **accept(n, proposal)** is sent by an proposer and is received by a acceptor.

5. **accepted(proposal)**. The message contains a single argument, *proposal*, which corresponds to the value that the acceptor has accepted.

   Message type **accepted(proposal)** is sent by an acceptor and is received by a learner.

b) Local variables for each type of agent in the Paxos algorithm:

1. **Proposer**. A local variable of proposer is *last_n*, which is a timestamp. *last_n* is the last value of n sent by this proposer in a **prepare(n)** message. The other local variable is *grantors*, which is a set of **granted(n, ...)** messages received by this proposer, used to track when the majority of acceptors are ready to accept a proposal. *grantors* is reset to an empty set every time the proposer sends out a new **prepare(n)** message and updates *last_n*. Initially *last_n* is some $None$ and *grantors* is an empty set.

2. **Acceptor**. Acceptors maintain three local variables. *highest_prepare_n* is a timestamp corresponding to the highest value of n acceptor has received through a **prepare(n)** message. *latest_accepted* and *latest_accepted_n* correspond to the latest accepted proposal and the timestamp of that proposal respectively. Said proposal must have been received through a (successful) **accept(n, proposal)** message. Initially *highest_prepare_n* is some arbitrary negative value, while *latest_accepted* and *latest_accepted_n* are both $None$.

3. **Learner**. Learner maintains two local variables. First variable is *learned*, which corresponds to the value of the proposal the learner has learned by receiving an **accepted(proposal)** message. The second variable is the hashmap *proposals*. The keys in this hashmap are the IDs of acceptors and the values are the proposals received from relevant agents through **accepted(proposal)** messages. Initially *learned* is $None$ and *proposals* is a hashmap with a value for each acceptor ID initialised to $None$.

c) Programs for each agent type in the Paxos algorithm written in (loose) UNITY notation can be found below. Note that in code any non-$None$ timestamp is greater than $None$. Additionally, if the message has been received by there is no action corresponding to it (e.g. the second part of guard was not satisfied), the message is ignored.

Program Proposer

var id : Unique ID of this agent

$last\_n$: (t, id)

$grantors$: set of received **granted(...)** messages

initially $last\_n = None$

$\wedge\ grantors = \varnothing$

actions

if *triggered*

then Pick some $n = (t, id)$ where t is higher than that in $last\_n$,
send prepare(n) message to a majority set of acceptors,
assign $last\_n, grantors := n, \varnothing$.

if receive $granted(n, latest\_accepted, latest\_accepted\_n) \wedge n = last\_n$

then add the received **granted(...)** message to the $grantors$ set.

if $grantors$ set contains messages from the majority of acceptors $\wedge$ $latest\_accepted = None$ in all messages

then pick some value *proposal* representing a desired proposal,
send out **accept(last_n, proposal)** message to acceptors appearing in $grantors$ set,
assign $grantors := \varnothing$.

if $grantors$ set contains messages from the majority of acceptors $\wedge$ $latest\_accepted \neq None$ in some messages

then search messages in $grantors$ to find the *proposal* corresponding to the highest $latest\_accepted\_n$,
send out **accept(last_n, proposal)** message to acceptors appearing in $grantors$ set,
assign $grantors := \varnothing$.

Program Acceptor

var $latest\_accepted$: proposal

$highest\_prepare\_n, latest\_accepted\_n$: (t, id)

initially $highest\_prepare\_n = latest\_accepted = latest\_accepted\_n = None$

actions

if receive $prepare(n) \wedge n > highest\_prepare\_n$

then respond with **granted(n, latest_accepted, latest_accepted_n)**,
assign $highest\_prepare\_n := n$.

if receive $prepare(n) \wedge n \leq highest\_prepare\_n$

then respond with **nack_prepare(n)**.

if receive $accept(n, proposal) \wedge n \geq highest\_prepare\_n$

then send out **accepted(proposal)** to all learners,
assign $latest\_accepted, latest\_accepted\_n := proposal, n$.

Program Learner

var $learned$: proposal

$proposals$: hashmap with acceptor IDs as keys and proposals as values

initially $learned = None$

$\wedge\ (\forall\, id : id \in \{\text{acceptor IDs}\} : proposals[id] = None)$

actions

if receive $accepted(proposal)$

then assign $proposals[id] := proposal,$ where id is ID of the sender.

if majority of values in $proposals$ are equal to some value $proposal \wedge proposal \neq None$

then assign $learned := proposal$.