|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Name of the algorithm** | **Network** | **msg** | **Send** | **Send has a receiver** | **Receiver has a sender** | **Observações** |
| **2pc.hny** | set | { .dst: req.src, .src: self, .response: "no" } | Adds a message to the network | Yes (dst) | No | Each bank and each coordinator is a thread.  The receive() method is used by coordinators in an atomically when exists statement to wait for a response from each bank involved in a transaction.  In the arguments of the methods is always the coordinator (self). |
| **abd.hny** | bag | { .type: .response, .dst: m.src, .value: (t, v) } | Adds a message to the network | No, because it’s broadcast | Yes  (uid) | For efficiency of model checking the servers are anonymous.  Because the servers are anonymous, they may end up sending the same exact message, but clients are waiting for a particular number of messages.  Each server also keeps track of all the requests its already received so it doesn’t handle the same request twice.  Read and write operations are both invoked with a unique identifier uid. Both start by broadcasting a .read request to all servers and then waiting for a response from N – F servers. The receive() function uses the bag.combinations method to find all combinations of subsets of responses of size N – F. The second phase of each operation is similar. |
| **abp.hny** | Channel  Pointer (pchan) | { .seq: !seq, .payload: payload }  { .ack: m.seq } | The message goes to the channel | Yes, if the channel is to and from a specific person | Yes, if the channel is to and from a specific person | We can model each channel as a variable that either contains a network message or nothing (we use () to represent nothing in the model). Let s chan be the channel from Alice to Bob and r chan the channel from Bob to Alice. net send(pchan, m) models sending a message m to !pchan, where pchan is either ?s chan or ?r chan. The method places either m (to model a successful send) or () (to model loss) in !pchan. net recv}(pchan) models checking !pchan for the next message. If there is a message, it sets !pchan to ().  method app send(net, seq, msg) retransmits { .seq: !seq, .payload: msg } until an acknowledgment for !seq is received.  Method app recv(net, seq) returns the next successfully received message (with the given sequence number bit) if any. |
| **bosco.hny** | bag | Tuple  (round, estimate) | Adds a message to the network | No  (Broadcast) | No | In this particular algorithm, all messages are broadcast to all processors, so they do not require a destination address. The N processors go through a sequence of rounds in which they wait for N – F messages, update their state based on the messages, and broadcast messages containing their new state. The reason that a processor waits for N – F rather than N messages is because of failures: up to F processors may never send a message and so it would be unwise to wait for all N. You might be tempted to use a timer and time out on waiting for a particular processor. But how would you initialize that timer? While we will assume that the network is reliable, there is no guarantee that messages arrive within a particular time. We call a set of N – F processors a quorum. A quorum must suffice for the algorithm to make progress |
| **bosco2.hny** | bag | Tuple  (round, estimate) | Adds a message to the network | No  (Broadcast) | No |  |
| **byzbosco.hny** | bag | Tuple  (round, proposal) | Adds a message to the network | No  (Broadcast) | No |  |
| **chain.hny** | set | - | Adds a message to the network | Yes  (dst) | Yes  (I think it’s the self) | It has a send to a specific replica, and it has a broadcast to send a msg to all replicas.  Each replica maintains its own history hist and a chain configuration config. The replica executes a loop in which it receives and atomically handles messages until it crashes. As before, one of the replicas cannot crash. Because replicas do not want to handle the same message twice, each replica maintains a set received of messages it has already handled. Each replica then waits for a message on the network it has not already handled before. |
| **leader.hny** | set | Tuple  (dst, id, found) | Adds a message to the network | Yes  (dst) | Yes  (id) | If a processor receives its own identifier, it knows its the leader. The Harmony code checks this using an assertion. In real code the processor could not do this as it does not know the identifier of the leader, but assertions are only there to check correctness. The processor then sends a message to its successor that the leader has been found. If the processor receives an identifier higher than its own, the processor knows that it cannot be the leader. In that case, it simply forwards the message. A processor stops when it receives a message that indicates that the leader has been identified.  When there are multiple messages to receive by a processor, every possible order is tried (including receiving the same message multiple times). |
| **lossy.hny** | It’s a channel (chan) | Penso que só possa ser “ping” ou “pong”, consoante seja para o servidor ou para o cliente. | Adds a message to the channel | No  (Or yes, if the channel has a specific sender and receiver) | No  (Or yes, if the channel has a specific sender and receiver) | In this case, we only have one client and one always active server. |
| **Needhamschroeder.hny** | set | Dictionary {.dst : p, .contents : m} | Adds a message to the network | Yes  (dst) | Yes  (.initiator) | It’s incorrect.  See explanation in the manual. |
| **paxos.hny** | bag | Tuple  (b,p,t,e) | Adds a message to the network | No | No | O send é broadcast. |
| **paxos3.hny** | It’s through channels [[],] | { .type: "p1a", .src: self, .ballot: leader\_ballot } | Adds a message to the channel | Yes  (p) | Yes (src : self, that is always going to be the leader) |  |
| **rsm.hny** | list (queue) | self | Adds a message to the network | No  (It ends up being like a broadcast) | Yes: the msg itself, because it has the self of the sender | In a replicated state machine, the abstract network maintains this history as an ordered queue of messages. NOPS clients each place an operation on the network. The replicas process messages from the ordered network. |
| **rsmspec.hny** | list (queue) | self | Adds a message to the network | No  (It ends up being like a broadcast) | Yes: the msg itself, because it has the self of the sender | What changes this one from the previous one is that this one has the history in a variable called hist. |
| **anonbosco.hny** | bag | Tuple  (round, estimate) | Adds a message to the network | No  (Broadcast) | No  (unless it’s related with round, but I don’t think so) |  |
| **anonbosco2.hny** | bag | Tuple  (round, estimate) | Adds a message to the network | No  (Broadcast) | No (unless it’s related with round, but I don’t think so) |  |
| **anonbosco3.hny** | bag | Tuple  (round, estimate) | Adds a message to the network | No  (Broadcast) | No (unless it’s related with round, but I don’t think so) |  |
| **benor.hny** | bag | Tuple  (r,p,e) ->  (round, phase, estimate) | Adds a message to the network | No  (Broadcast) | No  (unless it’s related with round, but I don’t think so) |  |

**To do:**

Ver mais como as bags, listas, tuplos e etc funcionam

Ver o algoritmo por alto, na wikki e tals

**Notas:**

Usam bastante como bibliotecas as funções que criam para a network, aka, send, receive, etc.

O chain parece ser mais à base daquilo que queremos.

Para termos quem envia e quem queremos que receba, provavelmente temos de colocar essa informação na msg ou então criar channels.

chainaction.hny tem operations e não tem msg. Não tem network. Não sei se será relevante para nós.