1 Olympus

1: **function** INITIALIZECONFIG(clients, t, h) ▷ Constructs a new configuration with new replicas. Distributes keys.
2: $self.publicKeys \leftarrow$ GENERATEPUBLICKEYS$(clients, t)$;
3: $self.clients \leftarrow clients$;
4: **for all** client in self.clients **do**
5: $keysMessage \leftarrow \{\}$;
6: $keysMessage.privateKey \leftarrow$ GENERATEPRIVATEKEY;
7: $keysMessage.publicKeys \leftarrow self.publicKeys$;
8: SEND$(client, keysMessage, self)$;
9: **end for**
10: $self.currentConfig \leftarrow \{\}$;
11: $self.currentConfig.t \leftarrow t$;
12: **for** $i \leftarrow 0, 2t$ **do**
13: $replicas \leftarrow replicas \cup$ CREATEREPLICA(i, h);
14: **end for**
15: $self.currentConfig.head \leftarrow replicas[0]$;
16: $self.currentConfig.tail \leftarrow replicas[|replicas| - 1]$;
17: $self.currentConfig.replicas \leftarrow replicas$;
18: INITIALIZETOPOLOGY;
19: **end function**
20:
21: **function** CREATEREPLICA(rid, h) ▷ Initialize a new Replica process.
22: $privateKey \leftarrow$ GENERATEPRIVATEKEY();
23: $replica \leftarrow$ **new***Replica*;
24: replica.INITIALIZE$(rid, h, privateKey, self.publicKeys)$;
25: **return** replica;
26: **end function**
27:
28: **function** INITIALIZETOPOLOGY ▷ Make replicas aware of the network topology.
29: $self.currentConfig.head.pre \leftarrow NULL$;
30: $self.currentConfig.tail.next \leftarrow NULL$;
31: **for** $i \leftarrow 1, |replicas| - 2$ **do**
32: $self.currentConfig.replicas[i].pre \leftarrow self.currentConfig.replicas[i-1]$;
33: $self.currentConfig.replicas[i].next \leftarrow self.currentConfig.replicas[i+1]$;
34: **end for**
35: **end function**
36:
37: **function** FETCHCONFIG ▷ Return the current configuration.
38: **return** self.currentConfig;
39: **end function**
40:
Require: A reconfiguration request is received.
41: **function** ONRECONFIGREQUEST(message, source) ▷ Handle a reconfiguration request.
42: **if** $\neg$ISVALIDSIGNATURE$(message, self.publicKeys[source])$ **then**

```
43:         return;
44:     else if ¬message.timeout&&¬isValidMisbehaviorProof(message)
    then
45:         return;
46:     end if
47:     sendWedgeRequests;
48: end function
49:
50: function sendWedgeRequests
51:     for replica in self.currentConfig.replicas do
52:         wedgeRequestMessage ← {};
53:         wedgeRequestMessage.client ← wedgeRequestMessage.client;
54:         signedMessage ← sign(wedgeRequestMessage, self.publicKeys[replica]);
55:         send(replica, signedMessage, self);
56:     end for
57:     quorumSatisfied ← |self.replies| == self.currentConfig.t + 1;
58:     wait(quorumSatisfied);
59: end function
60:
61: function isValidMisbehaviorProof(message)        ▷ validate a message
    containing a proof of misbehavior.
62:     if message.misbehaviorProof&&message.misbehaviorProof.orderProof
    then
63:         return ≠ isValidOrderProof(message, self.currentConfig.replicas, self.publicKeys);
64:     end if
65:     if message.misbehaviorProof&&message.misbehaviorProof.invalidResultProof
    then
66:         return¬isValidResult(message.misbehaviorProof.invalidResultProof);
67:     end if
68:     slotOperationPairs ← getSlotOperationPairs(message.h, message.conflictingSlot);
69:     return|slotOperationPairs| > 1;
70: end function
71:
72: function getSlotOperationPairs(h, slot)        ▷ Get the slot operation
    pairs (s, o) such that s = slot.
73:     slotOperationPairs ← {};
74:     for orderProof in h do
75:         if orderProof[ORDER_PROOF_SLOT_INDEX] == slot then
76:             slotOpPair ← (orderProof[SLOT_INDEX], orderProof[OPERATION_INDEX]);
77:             slotOperationPairs ∪ slotOpPair;
78:         end if
79:     end for
80: end function
81:
Require: A quorum of wedged requests is met
82: function onWedgedQuorumSatisfied(messages)        ▷ Handle the case
    when a quorum of wedged responses is satisfied.
83:     hist ← constructHistorySeed(messages);
84:     initializeConfiguration(self.clients, self.currentConfiguration.t, hist);
85: end function
```

```
86:
87: function sendCatchUp(messages)            ▷ Prepare and send catch up
    messages to all replicas in the quorum
88:     longestHistory ← getLongestHistory(message);
89:     for all message in messages do
90:         maxOperationSlot ← getMaxOperationSlotFromHistory(message.history);
91:         catchUpMessage ← {};
92:         catchUpMessage.deltaHistory ← deltaHistory;
93:         for all orderProof in longestHistory do
94:             if orderProof[SLOT_INDEX] > maxOperationSlot then catchUpMessage.deltaHistory ←
    catchUpMessage.deltaHistory ∪ orderProof;
95:             end if
96:         end for
97:         send(message.replica, catchUpMessage, self);
98:     end for
99: end function
100:
101: function getLongestHistory(messages)        ▷ From a list of correct
    wedged statements get the longest history.
102:     maxOperationSlot ← −∞;
103:     minSlot ← ∞;
104:     longestHistory ← {};
105:     for all message in messages do
106:         maxOperationSlotForReplica ← getMaxOperationSlotFromHistory(message.history);
107:         if maxOperationSlotForReplica > maxOperationSlot then
108:             longestHistory ← message.history;
109:         end if
110:     end for
111:     return longestHistory;
112: end function
113:
114: function getMaxOperationSlotFromHistory(history)    ▷ Returns
    the maximum slot value from a given a list of order proofs
115:     maxOperationSlot ← −∞;
116:     for all orderProof in message.history do
117:         if orderProof[ORDER_PROOF_SLOT_INDEX] > maxOperationSlot
    then
118:             maxOperationSlot ← orderProof[ORDER_PROOF_SLOT_INDEX];
119:         end if
120:     end for
121:     return maxOperationSlot;
122: end function
123:
Require: A caught up message has been received.
124: function onCaughtUpMessage(message, source)
125:     if source in self.quorumReplicas and isValidSignature(message.signature,
    self.publicKeys[source]) then
126:         self.caughtUpMessageCache[message.hash] ← self.caughtUpMessageCache[message.hash]∪
    source;
127:         for all entry in caughtUpMessageCache do
```

```
128:             if |entry.values| > t + 1 then
129:                 self.caughtUpMessageCache ← {};
130:                 self.quorumReplicas ← {};
131:                 INITHIST(entry.key)
132:             end if
133:         end for
134:     end if
135: end function
136:
Require: A wedged response has been received.
137: function ONWEDGEDRESPONSE(message, source)        ▷ Handle a single
     wedged response.
138:     if ¬ISVALIDSIGNATURE(message, self.publicKeys[source]) then
139:         return;
140:     end if
141:     for all orderProof in message.history do
142:         if ≠ ISVALIDORDERPROOF(message, self.replicas, self.publicKeys)
     then
143:             SENDWEDGEREQUESTS;
144:             return;
145:         end if
146:     end for
147:     self.replies ← self.replies ∪ message;
148:     if —self.replies— ¿= self.t + 1 then
149:         messages ← {};
150:         for all reply in replies do
151:             messages ← messages ∪ reply;
152:         end for
153:         SENDCATCHUP(sendCatchUp);
154:     end if
155: end function
```

2 client

```
1: function updateConfig  ▷ Fetches the configuration from Olympus and updates the local copy at a fixed interval
2:     self.currentConfig ← Olympus::fetchConfig;
3: end function
4:
Require: A keys message from Olympus has been received.
5: function onKeysMessage(message)
6:     self.privateKey ← message.privateKey;
7:     self.publicKeys ← message.publicKeys;
8: end function
9:
10: function issueRequest(operation)  ▷ Makes a request to head
11:     head ← self.currentConfig.head;
12:     requestMessage ← createRequest(operation);
13:     self.pendingRequests[requestMessage.operation] ← requestMessage;
14:     send(head, requestMessage, self);
15:     self.isResponseReceived ←self.pendingRequests[requestMessage.operation] == NULL;
16:     wait(self.isResponseReceived, TIMEOUT_DURATION);
17: end function
18:
19: function createRequest(operation)  ▷ Creates the request object
20:     requestMessage.content.client ← self;
21:     requestMessage.content.isRetry ← false;
22:     requestMessage.content.operation ← operation;
23:     requestMessage.signedHash ← sign(requestMessage.content, self.privateKey);
24: end function
25:
Require: A response is received
26: function onResultReceived(message)  ▷ Contains actions to be taken when a message is received
27:     if self.pendingRequests[message.content.operation] == NULL then
28:         return;
29:     else if isValidResult(message, self.currentConfig.replicas, self.publicKeys) then
30:         del self.pendingRequests[message.content.operation];
31:         processResult(message.content.operation, message.content.result);
32:     else
33:         retryRequest(pendingRequests[message.content.operation]);
34:     end if
35: end function
36:
Require: An error message is received
37: function onErrorReceived(message)  ▷ Contains actions to be taken when an error message is received
38:     retryRequest(self.pendingRequests[message.content.operation]);
39: end function
40:
```

Require: A request is timed out

41: **function** ONTIMEOUT(requestMessage) ▷ Contains actions to be taken on a timeout
42: RETRYREQUEST(pendingRequests[message.content.operation]);
43: **end function**
44:
45: **function** RETRYREQUEST(requestMessage) ▷ Handles retransmission of a failed request (timeout or incorrect response)
46: $requestMessage.isRetry \leftarrow true$;
47: $requestMessage.signedHash \leftarrow$ SIGN($requestMessage, self.privateKey$);
48: **for** replica in self.currentConfig.replicas **do**
49: SEND(replica, requestMessage);
50: **end for**
51: $self.isResponseReceived \leftarrow self.pendingRequests[requestMessage.operation] == NULL$;
52: WAIT(self.isResponseReceived, timeoutDuration);
53: **end function**
54:

3 Replica

```
1: function INITIALIZE(id, h, privateKey, publicKeys)   ▷ Initialize a replica.
2:     self.id ← id;
3:     self.h ← h;
4:     self.replicas ← OLYMPUS::FETCHCONFIG.replicas;
5:     self.privateKey ← privateKey;
6:     self.publicKeys ← publicKeys;
7:     self.state ← REPLICA_ACTIVE;
8:     self.resultCache ← {};
9:     return self;
10: end function
11:
Require: An update shuttle is received
12: function ONUPDATESHUTTLE(message, source)   ▷ Handle an incoming
    update shuttle
13:     if ¬ISVALIDSIGNATURE(message, self.publicKeys[source]) then
14:         return;
15:     end if
16:     if message.slot <= self.lastSignedSlot then
17:         return;                                   ▷ Drop the message.
18:     end if
19:     if ≠ ISVALIDORDERPROOF(message, self.replicas, self.publicKeys) then
20:         self.state ← REPLICA_IMMUTABLE;
21:         reconfigRequestMessage ← {};
22:         reconfigRequestMessage.misbehaviorProof ← {};
23:         reconfigRequestMessage.misbehaviorProof.orderProof ← message.orderProof;
24:         signedMessage ← SIGN(reconfigRequestMessage, self.privateKey)
25:         SEND(Olympus, signedMessage, self);
26:     else
27:         message.orderProof ← UPDATEORDERPROOF(message);
28:         self.history ∪ message.orderProof;
29:         self.lastSignedSlot ← message.slot;
30:         result ← UPDATERUNNINGSTATE(message.operation);
31:         message.resultProof ← UPDATERESULTPROOF(message, result);
32:         ROUTE(message, DIRECTION_FORWARD);
33:         WAIT(self.resultShuttleReceived);
34:     end if
35: end function
36:
Require: Replica waiting for result shuttle timed out
37: function ONRESULTSHUTTLETIMEOUT(message, source)   ▷ Handle a
    timeout for waiting on a result shuttle.
38:     reconfigRequestMessage ← {};
39:     self.state ← REPLICA_IMMUTABLE;
40:     signedMessage ← SIGN(reconfigRequestMessage, self.privateKey);
41:     SEND(Olympus, signedMessage, self);
42: end function
43: ▷ order statement: (slot, operation) Config omitted as there can be only 1
    active configuration
```

```
44: function updateOrderProof(message)  ▷ Update the order proof to include an order statement made by this replica.
45:     orderStatement ← (message.slot, message.operation, self);
46:     signedOrderStatement ← sign(orderStatement, self.privateKey);
47:     message.orderProof.orderStatements ∪ signedOrderStatement;
48:     message.orderProof.replica ← self;
49:     self.history ∪ message.orderProof;
50: end function
51:
52: function updateRunningState(operation)  ▷ Apply the operation specified on the designated object and return its value.
53:     return update(self.object, message.operation);
54: end function
55:
56: function getRunningState
57:     return get(self.object);
58: end function
59:
60: function updateResultProof(message, result)  ▷ Update the result proof to include a hash of the result computed at this replica.
61:     resultStatement ← (message.operation, hash(result));
62:     signedResultStatement ← sign(resultStatement, self.privateKey);
63:     message.resultProof ∪ signedResultStatement;
64: end function
65:
Require: A result shuttle is received
66: function onResultShuttle(message, source)  ▷ Handle an incoming result shuttle
67:     if ¬isValidSignature(message, self.publicKeys[source]) then
68:         return;
69:     end if
70:     if isValidResult(message, self.replicas, self.publicKeys) then
71:         resultShuttleReceived ← true;
72:         resultCache[message.operation] = message;
73:         forward(message, DIRECTION_BACKWARD);
74:     else
75:         handleInvalidResultProofError(message);
76:     end if
77: end function
78:
79: function handleInvalidResultProofError(message)  ▷ Handle a result proof error
80:     state ← REPLICA_IMMUTABLE;
81:     reconfigRequestMessage ← {};
82:     reconfigRequestMessage.misbehaviorProof ← {};
83:     reconfigRequestMessage.misbehaviorProof.invalidResultProof ← message.resultProof;
84:     signedMessage ← sign(reconfigRequestMessage, self.privateKey)
85:     send(Olympus, signedMessage, self);
86: end function
87:
```

Require: A wedge request is received from Olympus ▷ Handle an incoming wedge request.
88: **function** onWedgeRequest(message, source)
89: **if** ¬isValidSignature($message, self.publicKeys[source]$) **then**
90: **return**;
91: **end if**
92: $wedgedMessage \leftarrow \{\}$;
93: $wedgedMessage.replica \leftarrow self$;
94: $wedgedMessage.history \leftarrow self.history$;
95: $signedMessage \leftarrow$ wedgedMessage, self.privateKey(;)
96: send(Olympus, signedMessage, self);
97: **end function**
98:
Require: A request is received from the client
99: **function** onClientRequest(message, source) ▷ Handle an incoming client request.
100: **if** ¬isValidSignature($message, self.publicKeys[source]$) **then**
101: **return**;
102: **end if**
103: **if** $message.isRetry$ **then**
104: handleRetransmissionRequest($message, source$);
105: **else**
106: route($message, DIRECTION\_BACKWARD$);
107: **end if**
108: **end function**
109:
110: **function** handleRetransmissionRequest(message, client) ▷ Handle an incoming retransmission request.
111: **if** $state == REPLICA\_ACTIVE$ **then**
112: $resultMessage \leftarrow \{\}$;
113: $cachedResultShuttle \leftarrow resultCache[message.operation]$;
114: $resultMessage.result \leftarrow cachedResultShuttle.result$;
115: $resultMessage.operation; \leftarrow cachedResultShuttle.operation$;
116: $resultMessage.resultProof \leftarrow cachedResultShuttle.resultProof$;
117: send($client, resultMessage, self$);
118: **else**
119: ▷ Transmit an error message to the client as it has reached an immutable replica.
120: $errorMessage \leftarrow \{\}$;
121: $errorMessage.type \leftarrow ERROR_MESSAGE$;
122: send(message.client, errorMessage, self);
123: **end if**
124: **end function**
125:
Require: A catch-up message has been received.
126: **function** onCatchUp(message)
127: **if** ¬isValidSignature($message, self.publicKeys[Olympus]$) **then**
128: **return**;
129: **end if**
130: **for all** orderProof in message.deltaHistory **do**

```
131:         updateRunningState(orderProof.operation);
132:     end for
133:     caughtUpMessage ← {};
134:     runningState ← getRunningState(;)
135:     caughtUpMessage.ch ← hash(runningState);
136:     signedMessage ← sign(caughtUpMessage, self.privateKey);
137:     send(Olympus, signedMessage, self);
138: end function
139:
140: function route(message, direction)          ▷ route a shuttle upstream or
     downstream.
141:     if direction == DIRECTION_FORWARD then
142:         send(self.next, message, self);
143:     else
144:         send(self.pre, message, self);
145:     end if
146: end function
147:
148: function receiveCheckpointShuttle(checkpointShuttle) ▷ Actions to
     be taken when a checkpoint is received
149:     if isValidCheckpoint(checkpointShuttle) then
150:         if checkpointShuttle.content.isComplete then
151:             truncateHistory
152:             route(checkpointShuttle, DIRECTION_BACKWARD);
153:             self.checkpointInitiated ← false;
154:             cleanResultCache
155:         else
156:             checkpointShuttle.checkpointProof ∪ createCheckpoint;
157:             self.CHECKPOINT_SLOT_ID ← checkpointShuttle.checkpoint;
158:             self.checkpointInitiated ← true;
159:             route(checkpointShuttle, DIRECTION_FORWARD);
160:             wait(checkpointInitiated == false, timeoutDuration);
161:         end if
162:     end if
163: end function
164:
165: function createCheckpoint                  ▷ Creates a checkpoint
166:     checkpoint.content.hash ← hash(self.object);
167: end function
168:
169: function sendResult(resultProof, result)          ▷ Sends result of the
     operation to the client
170:     resultObject ← {};
171:     resultObject.resultProof ← resultProof;
172:     resultObject.result ← message.result;
173:     resultObject.hash ← cryptographicHash(resultObject.result);
174:     resultObject.operation ← message.operation;
175:     send(message.client, resultObject, self)
176: end function
177:
```

178: **function** TRUNCATEHISTORY
179: ▷ Truncate all but latest $MIN\_ORDER\_HISTORY\_SIZE$ number of records from $self.history$;
180: **end function**
181:
Require: Replica waiting for checkpoint shuttle timed out
182: **function** ONCHECKPOINTSHUTTLETIMEOUT ▷ Handle a timeout for waiting on a checkpoint shuttle.
183: $reconfigRequestMessage \leftarrow \{\}$;
184: $self.state \leftarrow REPLICA\_IMMUTABLE$;
185: $reconfigRequestMessage.signedHash \leftarrow$ SIGN$(reconfigRequestMessage, self.privateKey)$
186: REQUESTRECONFIGURATION$(reconfigRequestMessage)$;
187: **return**;
188: **end function**
189:
190: **function** CLEANRESULTCACHE ▷ Removes operation older than the caching threshold from the resultCache
191: **for** $\forall Map[operation, resultProof] \in resultCache$ **do**
192: **if** operation older than OPERATION_CACHING_THRESHOLD **then**
193: del $map[operation]$;
194: **end if**
195: **end for**
196: **end function**
197:

4 Head(Extends Replica)

Require: A request is received

function onClientRequest(message) ▷ Actions taken when a request is received from a client
  if $processedRequests[message.content.operation] != NULL$) **then**
    return $processedRequests[message.content.operation]$;
  else if $pendingRequests[message.content.operation] != NULL$ **then**
    $isResponseReceived \leftarrow pendingRequests[message.content.operation] == NULL$;
    wait(isResponseReceived, TIMEOUT_DURATION);
  else
    processRequest(message)
  end if
end function

function processRequest(message)
  if ¬isValidSignature($message, self.publicKeys[message.content.client]$) **then**
    return;
  end if
  if $message.content.operationType == READ$ **then**
    return handleRead(message);
  else if $message.content.operationType == UPDATE$ **then**
    return handleUpdate(message);
  else
    return;
  end if
end function

function handleUpdate(updateMessage) ▷ Actions to be taken when an update operation is requested
  $slot \leftarrow$ getSlot;
  $message \leftarrow$ createShuttle($updateMessage, slot$);
  $message.orderProof \leftarrow$ updateOrderProof($message$);
  $r \leftarrow$ updateRunningState($self.object, updateMessage.operation$);
  $message.resultProof \leftarrow$ updateResultProof($message.resultProof, r$);
  route(message, DIRECTION_FORWARD);
end function

function handleRead(message) ▷ Actions to be taken when a read operation is requested
  if $processedRequests[message.content.operation] != NULL$) **then**
    return $processedRequests[message.content.operation]$;
  else
    return
  end if
end function

function createShuttle(request, slot) ▷ Creates a shuttle from the

request and slot
    $message \leftarrow \{\}$;
    $message.content.slot \leftarrow slot$;
    $message.content.operation \leftarrow request.content.operation$;
    $message.content.orderProof \leftarrow \{\}$;
    $message.content.resultProof \leftarrow \{\}$;
    $message.content.client \leftarrow request.content.client$;
end function

function onCheckpointTrigger ▷ Initiates checkpoint after every CHECKPOINT_TRIGGER_SIZE slots
    $checkpointShuttle \leftarrow$ createCheckpointShuttle;
    $self.checkpointInitiated \leftarrow true$;
    $checkpointShuttle.signedHash \leftarrow$ sign($checkpointShuttle, self.privateKey$);
    route(checkpointShuttle, DIRECTION_FORWARD);
    wait(checkpointInitiated == false, timeoutDuration);
end function

function createCheckpointShuttle ▷ Creates a checkpoint
    $checkpointShuttle \leftarrow \{\}$;
    $checkpointShuttle.content.checkpointProof \leftarrow \{\}$;
    $checkpointShuttle.content.isComplete \leftarrow false$;
    $checkpointShuttle.content.checkpoint \leftarrow self.CHECKPOINT\_SLOT\_ID + self.CHECKPOINT\_TRIGGER\_SIZE$;
    $checkpointShuttle.content.checkpointProof \cup$ createCheckpoint;
end function

function receiveCheckpointShuttle(checkpointShuttle) ▷ Actions to be taken when a checkpoint is received
    if isValidCheckpoint(checkpointShuttle) **then**
        if checkpointShuttle.content.isComplete **then**
            truncateHistory
            $self.checkpointInitiated \leftarrow false$;
            cleanResultCache
        else
            $checkpointShuttle.checkpointProof \cup$ createCheckpoint;
            $self.CHECKPOINT\_SLOT\_ID \leftarrow checkpointShuttle.checkpoint$;
            $self.checkpointInitiated \leftarrow true$;
            route(checkpointShuttle, DIRECTION_FORWARD);
            wait(checkpointInitiated == false, timeoutDuration);
        end if
    end if
end function

5 Tail(Extends Replica)

Require: A shuttle is received

function onShuttleReceived(message, source) ▷ Handle an incoming result shuttle
  if ¬isValidSignature($message, self.publicKeys[source]$) **then**
    return;
  end if
  if $message.slot <= self.lastSignedSlot$ **then**
    return; ▷ Drop the message.
  end if
  if hasSignatureMismatch($message.orderProof$) **then**
    handleOrderStatementSignMismatchError($message$);
  else if hasOrderConflict($message$) **then**
    handleOrderConflictError($message, source$);
  else
    $message.orderProof \leftarrow$ updateOrderProof($message$);
    $self.history \cup message.orderProof$;
    $self.lastSignedSlot \leftarrow message.slot$;
    $result \leftarrow$ updateRunningState($message.operation$);
    $message.resultProof \leftarrow$ updateResultProof($message, message.result$);
    $self.resultCache[message.clientId][message.operation] = message$;
    sendResult(message.resultProof, result);
    route($message, DIRECTION\_BACKWARD$);
  end if
end function

6 Commons

```
1: function isValidSignature(message, publicKey)  ▷ Apply the public key to the message's signed hash of the content and check if the result equals the message's content.This is called by all message handlers within the distributed system for checking message authenticity.
2:     decryptedContent ← decrypt(message.signedContent, publicKey);
3:     return decryptedContent == message.content;
4: end function
5:
6: function hasSignatureMismatch(orderProof, source)  ▷ Check for any order statement signature invalidity.
7:     for all signedOrderStatement in orderProof.orderStatements do
8:         if ¬isValidSignature(signedOrderStatement, self.publicKeys[orderProof.replica]) then
9:             return true;
10:        end if
11:    end for
12:    return false;
13: end function
14:
15: function isValidOrderProof(message, replicas, publicKeys)  ▷ Check the validity of order proofs.
16:    expectedHash ← hash(message.orderProof);
17:    validOrderProof ← true;
18:    replicasClone ← clone(replicas);
19:    for all orderStatement in message.orderProof.orderStatements do
20:        validOrderProof ← validOrderProof&&expectedHash == orderStatement.hash;
21:        pkey ← self.publicKeys[orderStatement.replica];
22:        validOrderProof ← validOrderProof&&isValidSignature(orderStatement, pkey);
23:        replicasClone ← replicasClone \ orderStatement.source;
24:    end for
25:    validOrderProof ← validOrderProof&&|replicasClone| == 0;
26:    return validOrderProof&&¬hasOrderConflict(message);
27: end function
28:
29: function hasOrderConflict(message)  ▷ Check if there is a conflicting operation for a particular client and slot.
30:    for all orderStatement in message.orderProof.orderStatements do
31:        if orderStatement.slot! = message.slot||orderStatement.operation! = message.operation then
32:            return true;
33:        end if
34:    end for
35:    return false;
36: end function
37:
38: function isValidResult(message, replicas, publicKeys)  ▷ Checks the validity of the response
39:    expectedHash ← hash(message.result);
```

```
40:     validResult ← true;
41:     replicasClone ← CLONE(replicas);
42:     for all resultStatement in message.resultProof do
43:         validResult ← validResult&&expectedHash == resultStatement.hash;
44:         pkey ← self.publicKeys[resultStatement.replica];
45:         validResult ← validResult&&ISVALIDSIGNATURE(resultStatement, pkey);
46:         replicasClone ← replicasClone \ resultStatement.source;
47:     end for
48:     validResult ← validResult&&|replicasClone| == 0;
49:     return validResult;
50: end function
51:
52: function SIGN(message, privateKey)          ▷ Computes an internal hash
    of a message and computes the signed hash of the message using the given
    private key. Called before sending any message in the system.
53: end function
54:
55: function DECRYPT(message, publicKey)       ▷ Applies a public key to the
    signed hash within a message.
56: end function
57:
58: function PROCESSRESULT(operation, result)      ▷ Pass the result for the
    corresponding operation to the application.
59: end function
60:
61: function GENERATEPUBLICKEYS(clients, t)     ▷ Generate public keys for
    each client and t no of keys for each replica and one for Olympus.
62: end function
63:
64: function GENERATEPRIVATEKEY                 ▷ Generate a private key.
65: end function
66:
67: function UPDATE(object, operation)     ▷ Applies the operation to a given
    object and returns the state of the object after the operation is applied.
68: end function
69:
70: function GET(object)                    ▷ Get the state for the specified object.
71: end function
72:
73: function WAIT(conditionVariable, timeoutDuration)          ▷ Sets up a
    timer internally with the given duration and begins a blocking wait for the
    conditionVariable to become true.
74: end function
75:
76: function SEND(destination, message, source)   ▷ Sends the message to the
    destination from the source.
77: end function
78:
79: function HASH(message)                  ▷ Constructs a hash of a statement.
80: end function
```