Hybster
A Highly Parallelizable Protocol for Hybrid Fault-Tolerant Service Replication

Johannes Behl  
TU Braunschweig  
behl@ibr.cs.tu-bs.de

Tobias Distler  
FAU Erlangen-Nürnberg  
distler@cs.fau.de

Rüdiger Kapitza  
TU Braunschweig  
rrkapitz@ibr.cs.tu-bs.de

April 21st, 2017
v0.1
## Contents

1 Introduction .............................................. 1

2 Notation ................................................ 1
   2.1 Modules and Operations ............................ 1
   2.2 Module Interaction ................................ 3
   2.3 Additional Expressions and Types ............... 6

3 System Model ........................................... 8
   3.1 Processes .......................................... 8
   3.2 Time .............................................. 10
   3.3 Communication ................................... 11
   3.4 Cryptography ..................................... 11
   3.5 Faults ............................................ 13

4 Specification of Hybster ............................... 15
   4.1 System Properties ................................ 15
   4.2 Auxiliary Modules ................................ 17
      4.2.1 Adaptive Timeouts ............................ 18
      4.2.2 Connections .................................. 19
   4.3 TrInX ............................................. 21
   4.4 Processes ......................................... 22
      4.4.1 Clients ....................................... 22
      4.4.2 Replicas ...................................... 22
   4.5 Invocation Protocol ................................ 24
      4.5.1 Client Side ................................... 24
      4.5.2 Replica Side ................................ 25
   4.6 Ordering Protocol ................................ 27
   4.7 Execution .......................................... 31
   4.8 Checkpointing Protocol ......................... 32
   4.9 View-Change Protocol ......................... 35
   4.10 State Transfer Protocol ...................... 41
1 Introduction

Hybster is a replication protocol for providing fault-tolerant services based on the state-machine approach [13]. It is designed for partially synchronous systems [9]. It does not rely on timing assumptions to ensure safety but is only guaranteed to make progress in phases where unknown upper bounds for message delivery and message processing exist. Moreover, using a hybrid fault model in which some components are trusted to only fail by crashing even when other components behave arbitrarily faulty [12, 14], Hybster requires only $2f + 1$ service replicas to tolerate up to $f$ faulty ones.

While the original paper that introduces Hybster [3] mainly presents the rationale behind the basic protocol and its parallelized variant, this technical report provides supplemental material such as a comprehensive system model description and formal specification of the protocol. The present document is the first version of the report that is planned to be updated once additional parts are ready. All published versions will be accessible at [2].

2 Notation

The formal specification of Hybster is essentially based on I/O automata [11] as employed by Castro and Liskov to specify the originator of Byzantine fault-tolerant service replication, namely PBFT [5, 6, 8]. However, the model used here differs from I/O automata mainly in two aspects. Firstly, it supports the direct interaction of modules where module instances explicitly trigger operations on other module instances. This makes dependencies between modules more apparent than the indirect interaction scheme based on operation signatures as used by the original I/O automata. Moreover, it allows more flexible interaction patterns between modules, especially helpful for describing the internal structure of complex processes. Secondly, to improve readability, the notation used in this document is inspired by pseudo-code notations like [4].

2.1 Modules and Operations

The specification of Hybster is given as a set of modules encapsulating state and defining operations on that state. Compared to an I/O automaton, a module is a more abstract concept. A module is not only a concurrent entity but allows for both asynchronous and synchronous operations. Furthermore, modules can be instantiated, there can be multiple identifiable instances of one and the same module. Instances share the behavior as described by the operations of the module but possess independent memory for state variables. Thus, modules and module instances have strong similarities to classes and objects of the object-oriented programming paradigm [1].

Tasks and Methods. Asynchronous operations are called tasks and are triggered according to a specified precondition. If the precondition of a task is satisfied for a module instance, the task of this instance is enabled and all enabled tasks are assumed to be executed asynchronously at some time, in a nondeterministic order, although only one task at a time. Tasks are classified as input, output, or internal operation. While tasks are not allowed to yield a direct return value, output tasks may specify a list of output parameters that serve as inputs for other tasks. Likewise, input tasks define a list of expected input parameters. Internal tasks can possess a parameter list as well. These parameters are neither inputs nor outputs and their only purpose is to make the function, the logical result of the internal tasks explicit.
Synchronous operations, called *methods*, are directly executed when invoked and may return a direct outcome. Methods are also classified as input, output, and internal operation but can be additionally declared as public. A public method is permitted to alter the state of the invoked module instance and yield a direct result; a public method can be regarded as both input and output. Besides tasks and methods, *global functions* can be specified. Such operations are not bound to instances of modules thus they cannot access any instance state.

**Figure 1:** Example for module definitions.

```plaintext
1 module NumberProducer
2 upon init() do
3     out_p ∈ (N)* := []
5 upon internal task produce(val ∈ N)
6     with
7     |out_p| < 10
8     val = random(0, 100)
9     do
10     out_p.append(val)

12 module NumberConsumer
13 upon init() do
14     sum_c ∈ N := 0
16 upon internal call consume(val ∈ N) do
17     sum_c := sum_c + val
```

Figure 1 gives an example for the definition of two modules. The module *NumberProducer* generates random numbers within an internal task *produce* and *NumberConsumer* summarizes numbers within an internal method *consume*. Both modules are not yet connected. How they can interact is described in the next section. The freely chosen index `p` in “module NumberProducer,” identifies an instance of the *NumberProducer* in the context of the module definition. The special method *init* is used to initialize a newly created instance of a module. “out_p ∈ (N)* := []” in Line 3 defines a single variable `out` for a new instance of the module *NumberProducer* denoted with `p`. `out_p` is an instance (∈) of a list ((...)*) of natural numbers (N) that is initially assigned (=) to an empty list ([])_. The definition of the *produce* task starts in Line 5. It is not allowed to have a direct return value. However, the parameter `val` is a free variable that is eventually bound to the value *produce* generates. `val` cannot be used directly. At this point, its only purpose is to make the function of *produce*, its logical outcome explicit. The *with* keyword starts the precondition section of the task definition. *produce* is only enabled when the list `out_p` contains fewer than 10 elements. Since `val` is a free variable, binding `val` to some randomly chosen natural number by “val = random(0, 100)” is always satisfied. In this example, `random` is assumed to be a global function defined within some other module. When a task is enabled, eventually all imperative statements in the section introduced by the keyword *do* will be executed in the sequence as specified. *produce* comprises only a single statement. “out_p.append(val)” adds the value bound to the variable `val` to the end of the list referenced by `out_p`. In sum, *produce* appends randomly chosen numbers to the list `out_p` as long as `out_p` currently contains fewer than 10 elements. The definition of *NumberConsumer* starting in Line 12 uses `c` to refer to an instance of that module. Instances of *NumberConsumer* possess one member variable `sum_c` that is a natural number initialized with 0. Further they possess one internal method *consume* that takes a value as argument and adds the value to `sum_c`.

**Operation Definition and Declaration.** The complete syntax for the definition of operations can be found in Figure 2. The keyword *upon* begins the definition of a task or a method. It is followed by an access modifier and an optional declaration of the operation type (*task* for tasks and *call* for methods). Operations may define a list of parameters. In the case of output and internal tasks, the parameters are output, in all other cases input parameters. Methods
may additionally define a direct return parameter, although this is optional. Access modifier, type declaration, name, and the list of all parameters form the signature of an operation. It follows a number of sections that constitute the body of the operation definition. Assertions are predicates that are expected to be true, they are assumptions that shall be made explicit. Assertions can be declared within the asserts section or anywhere else following an assert. A task must specify a set of preconditions that determine when the task is enabled and thus eligible for execution. Preconditions are given in the with section. Single preconditions can be prefaced by an optional if and if multiple conditions are specified, they are implicitly combined by a logical AND operation (\(\land\)). If an enabled task is executed or if a method is invoked, the sequence of statements within the do section is evaluated. Unless the operation is a global function, statements are allowed to modify the state of the invoked module instance. If a return statement is reached, the evaluation of the operation ends. For methods with return parameter, the return keyword has to be followed by an expression that specifies the return value. In conjunction with or as replacement for statements that explicitly alter state or define the values of output and return parameters, a set of postconditions may be declared that have to be satisfied after the evaluation of an operation. Postconditions are located in the yields section.

Figure 3: Syntax for operation declarations.

(a) Declare tasks.
1 declare <internal|input|output> task taskName(arg_0 \in T_0, \ldots, arg_{n-1} \in T_{n-1})
2
(b) Declare methods and global functions.
1 declare <internal|input|output|public|global> call methodName(arg_0 \in T_0, \ldots) \rightarrow ret \in R
Unidirectional Interaction. For example, Figure 4 shows two forms of a direct unidirectional interaction. Generally, in an unidirectional interaction, the output of one module, the source, is taken as input for another, the sink. Figure 4a presents the source-linked variant of this interaction scheme. In this example, two modules are defined, LinkingNumberSource and NumberSink. The module LinkingNumberSource is derived from the module NumberProducer and the current instance of this module is referred to as p. LinkingNumberSource inherits from NumberProducer the instance variable outp maintaining the list of generated but not yet consumed random numbers and the internal task produce. Additionally, it defines the instance variable snkp initialized with a new instance of the module NumberSink and an internal task transmit. transmit is enabled when outp contains at least one generated number. Upon being executed, transmit removes the first value from the list outp (outp.dequeue()) and invokes the input method receive of the instance snkp with this value as argument. The method invocation is synchronous, that is, all of its effects take place immediately. While being optional, the preceding keyword output shall emphasize this particular interaction between LinkingNumberSource and NumberSink. The module NumberSink is derived from NumberConsumer and thus inherits the variable sumc and the internal method consume, with c denoting the current module instance. The input method receive of NumberSink simply invokes consume of its base module handing over the given value val as argument. Taken all together, the task produce defined by NumberProducer asynchronously generates random natural numbers and places them in the list outp as long as this list contains fewer than 10 values. The task transmit asynchronously removes values from this list and delivers them as input to the NumberSink instance snkp. The invoked input method receive of snkp calls the internal method consume that finally adds the given value to the instance variable sumc, with c being the internal reference of the module instance also referenced by snkp.

Taken a source-linked interaction, the source is bound to the type of the sink and the source
instance needs a reference to the instance of the sink. In a sink-linked interaction scheme, this relation is reversed. The example depicted in Figure 4b resembles the previous one. The module \textit{NumberSource} extends \textit{NumberProducer} and defines the operation \textit{transmit} whereas the module \textit{LinkingNumberSink} extends \textit{NumberConsumer} and defines the operation \textit{receive}. However, in this case, \textit{transmit} is an output task that delivers the next random value from the list \textit{out}_p by means of the output parameter \textit{val}. The addressee of this value is not known to the \textit{NumberSource} module. Instead, instances of \textit{LinkingNumberSink} maintain a reference \textit{src}_c to a \textit{NumberSource} instance and their internal \textit{receive} task is bound to \textit{src}_c.\textit{transmit}, that is, \textit{receive} is directly called after \textit{src}_c.\textit{transmit} has been executed. Analogous to the \textbf{output} keyword, the \textbf{input} keyword emphasizing the interaction between \textit{NumberSource} and \textit{LinkingNumberSink} may be omitted. While receiving operations in sink-linked interactions are allowed to wait for multiple output events from different instances or even different output operations, they must not specify additional preconditions that would prevent any of their awaited outputs from being executed. In other words, sink-linked interactions must remain input-enabled as required from I/O automata [11].

\textbf{Bidirectional Interaction.} Whereas unidirectional interaction schemes link outputs to inputs, \textit{method invocations} are a form of \textit{bidirectional interaction} in which a callee takes an input and directly returns an output. Figure 5a presents an example. Again, \textit{NumberProducer} and \textit{NumberConsumer} from Figure 1 are extended to create interacting modules. The resulting modules \textit{IncreasingSum} and \textit{Accumulator} resemble the modules \textit{LinkingNumberSource} and \textit{NumberSink} of the source-linked interaction example presented in Figure 4a. Opposed to the input method \textit{receive} of \textit{NumberSink}, the public method \textit{add} of \textit{Accumulator} does not only add the given value to the variable \textit{sum}_c but also returns the resulting sum as direct outcome of the method invocation. To emphasize the invocation at the caller’s side, the keyword \textbf{invoke} can be used.

\begin{figure}[h]
\centering
\begin{tabular}{ll}
(a) Method invocation. & (b) Global function. \\
1 & 1 module \textit{Computer}_c \\
2 & 2 \textbf{upon global call} \textit{thinkDeep}() $\rightarrow \mathbb{N}$ do \\
3 & 3 \textbf{return} 42 \\
4 & 5 module \textit{Being}_b \\
5 & 6 \textbf{upon init()} do \\
6 & 7 \quad \textit{answer}_b \in \mathbb{N} \cup \{\emptyset\} := \emptyset \\
7 & 9 \textbf{upon internal task} \textit{askUltimate}() with \\
8 & 10 \quad \textit{answer}_b := \textit{Computer}::\textit{thinkDeep}() \\
9 & 13 \textit{do} \\
10 & 12 \quad \textit{answer}_b := \emptyset \\
11 & 11 \quad \textit{answer}_b := \textit{Computer}::\textit{thinkDeep}() \\
12 & 10 \\
13 & 9 \\
\end{tabular}
\end{figure}

\begin{figure}[h]
\centering
\begin{tabular}{ll}
1 module \textit{IncreasingSum}_p \\
2 \textbf{extends} \textit{NumberProducer} \\
4 \textbf{upon init()} do \\
5 \quad \textit{acc}_p := \textit{Accumulator}() \\
6 \quad \textit{sum}_p \in \mathbb{N} := 0 \\
8 \textbf{upon internal task} \textit{transmit}() with \\
9 \quad \textit{do} \\
10 \quad \quad \textit{[out}_p] > 0 \\
11 \quad \quad \textit{do} \\
12 \quad \quad \quad \textit{sum}_p := \textbf{invoke} \textit{acc}_p.\textit{add}(\textit{out}_p.\text{dequeue}()) \\
14 \textbf{module} \textit{Accumulator}(\textit{NumberConsumer})_c \\
15 \textbf{upon public call} \textit{add}(\textit{val} \in \mathbb{N}) $\rightarrow \textit{sum} \in \mathbb{N}$ do \\
16 \quad \textit{consume}(\textit{val}) \\
17 \quad \textbf{return} \textit{sum}_c \\
\end{tabular}
\end{figure}

\textbf{Global Functions.} Global functions share the syntax with methods but are not bound to any instance. While being specified within some module, they can be called from anywhere. Fig-
ure 5b illustrates the usage. The function \texttt{thinkDeep} is defined within the module \texttt{Computer} and simply returns 42 as constant. It is called by the module \texttt{Being} via “\texttt{Computer::thinkDeep}()”. Stating the defining module explicitly is, however, optional.

**Inner Modules.** Another form of interaction that leads to a very tight coupling between modules is the use of inner modules. An instance of an inner module possesses an implicit reference to the module instance that creates it and has access to all instance variables and operations be it internal or external of the referenced outer module instance. Figure 6 gives an example. Instances of the module \texttt{OuterNumberSource} create an instance of the inner module \texttt{InnerNumberSink}. The inner module instance thereby gets access to the list \texttt{out\_c} of produced random numbers that \texttt{OuterNumberSource} inherits from \texttt{NumberProducer}. If \texttt{out\_c} contains values, the internal task \texttt{consumeInner} of \texttt{InnerNumberSink} is enabled. When executed, the inner instance removes a value from \texttt{out\_c} maintained in the outer instance and passes it as argument to the \texttt{consume} method \texttt{InnerNumberSink} inherits from \texttt{NumberConsumer}.

\begin{figure}[h]
\centering
\begin{lstlisting}
1 module OuterNumberSource, 
2 extends NumberProducer

4 upon init() do 
5 \textit{snk\_p} := OuterNumberSource\_p::InnerNumberSink()

7 module OuterNumberSource::InnerNumberSink, 
8 extends NumberConsumer

10 upon internal task \textit{consumeInner}(val \in \mathbb{N}) 
11 with 
12 |\textit{out\_c}| > 0 
13 do 
14 \textit{consume}(\textit{out\_c}.\texttt{dequeue}())
\end{lstlisting}
\caption{Inner modules.}
\end{figure}

2.3 Additional Expressions and Types

While modules and their various ways of interaction are related to the general architecture of a system, the purpose of operations is to specify concrete algorithms by declaring pre- and postconditions or by defining them step-by-step as sequence of actions. This section gives an overview of expressions and basic types used to form predicates and imperative statements for the definition of operations.

**Sets.** Sets as unordered collections of objects are denoted by curly brackets. For example, \{1, 2, 3\} defines the set of the natural numbers 1, 2, and 3. The empty set can be written as \{\} or \emptyset. Concrete sets are instances of the basic type \texttt{Set}. \texttt{X \in Set} declares an arbitrary set \texttt{X}, and \texttt{X := \{1, 2, 3\}} assigns the set \{1, 2, 3\} to it. Common set operations are determining the union (\texttt{union}) or the difference (\texttt{difference}) of two sets. For example, \texttt{X := X \setminus \{2\} \cup \{4\}} results in \texttt{X = \{1, 3, 4\}}. The power set of some set \texttt{Y} is given by \texttt{P(Y)}. This can be used to specify a typed set: \texttt{Z \in P(N)} declares \texttt{Z} as a set of natural numbers.

**Tuples.** The ordered counterparts of sets are tuples which are instances of the basic type \texttt{Tuple}. Given \textit{n} objects \textit{x}_{0} to \textit{x}_{\textit{n}-1}, the corresponding tuple is written as \langle \textit{x}_{0}, \ldots, \textit{x}_{\textit{n}-1} \rangle. For
example, $t \in \textit{Tuple} := (1, 2, 3)$ defines $t$ as the tuple comprising the elements 1, 2, and 3 in that order. $t$ could have been also declared as $t \in \mathbb{N} \times \mathbb{N} \times \mathbb{N}$ or $t \in \mathbb{N}^3$, additionally specifying the types of the tuple’s elements.

**Lists.** While tuples are immutable, lists are ordered collections that can be modified. Lists are instances of the basic type $\textit{List}$ that is also denoted by $(\ldots)^*$. For example, a list $l$ of natural numbers can be declared with $l \in (\mathbb{N})^*$. Lists are given within square brackets. $l := [1, 2, 3]$ initializes $l$ with the elements 1, 2, and 3. The size of $l$ is $|l| = 3$. The type $\textit{List}$ defines the methods $\text{append}$, $\text{pop}$, and $\text{dequeue}$. $\text{append}$ adds an object to the end of the list, $\text{pop}$ returns and removes the last element and $\text{dequeue}$ the first element. Taken the list $l$ for example, $l$\text{append}(4) results in $l = [1, 2, 3, 4]$, $x := l$\text{pop()} in $x = 3 \land l = [1, 2]$, and $x := l$\text{dequeue()} in $x = 1 \land l = [2, 3]$.

**Booleans.** Classical truth values are represented by the basic type $\textit{Bool} \equiv \{\textit{True}, \textit{False}\}$, with $\textit{True} \equiv \emptyset = \emptyset$ and $\textit{False} \equiv \emptyset \neq \emptyset$. Based on that, a predicate can be defined as a method or function returning a value of type $\textit{Bool}$: $\text{upon } \ldots \text{ call predicateName}(\ldots) \rightarrow \textit{Bool}$. An alternative notation for $\textit{True}$ is $\top$ and for $\textit{False}$ $\bot$.

**Ellipses.** Predicates may take several input parameters. Sometimes it is sufficient to verify that a predicate holds for a subset of these parameters in conjunction with any binding of the remaining ones. For example, given a ternary predicate $P$ and two particular arguments $x$ and $y$, the question could be if $P$ is fulfilled for $x$ and $y$ and any additional argument $z$. To simplify the resulting expression $\exists z P(x, y, z)$, the elliptic form $P(x, y, \cdot)$ can be used. The extended variant for multiple arbitrarily chosen input parameters is written as $P(x, \ldots) \equiv \exists y \exists z P(x, y, z)$.

**None.** At some occasions it can be helpful to bind a variable to a special value that resides outside the main domain of that variable, for instance, to denote that the variable is unset and does not contain any real value. As such special value $\textit{None}$ may be used, also denoted by $\emptyset$. With $x \in \mathbb{N} \cup \{\emptyset\}$, the variable $x$ can reference a natural number or the special value $\emptyset$.

In addition to basic types, specifications can make use of common control flow structures such as conditionals and loops but also of more specific structures with regard to asynchronous operations.

**Sequences of Statements.** Sequences of statements are usually given as list, each statement in a single line at the same level of indentation. Figure 7a shows an example in which the values of two variables $x$ and $y$ are swapped using an additional variable $\textit{tmp}$. An alternative notation of this example where statements are written in a single line separated by semicolons is given in Figure 7b. If single statements are wrapped, be it for clarity or because they do not fit into the space for a single line, overhanging parts have to be indented by two levels. As Figure 7c illustrates, this can happen within a single line or by splitting the statement such that it spans multiple lines.

**Conditionals and Loops.** The notation also supports typical control flow statements for conditional branches or repetitive executions. Figure 8a gives an example in which two variable $x$ and $y$ are compared and a return value is chosen depending on the outcome of this comparison. If a condition is satisfied, the associated block indented by one level is executed. Although these
blocks contain only a single \texttt{return} in this example, blocks can generally span an arbitrary number of statements. An example for a loop that iterates over all elements of a set \(X\) and passes them to a procedure \texttt{processX} is depicted in Figure 8b.

### Synchronous Task Evaluations

As described in Section 2.1, tasks are enabled when their specified precondition is satisfied but they are only executed asynchronously without any guaranteed order. Furthermore, if the precondition of a task \(t\) depends on state that is altered by other tasks such that \(t\) is enabled and disabled depending on which other tasks are executed, there is no guarantee that the task \(t\) is executed at all. To support occasions in which a task is to be executed synchronously if necessary, the \texttt{check} keyword is provided. For example, if \texttt{"check someTask()"} is called, the precondition of task \texttt{someTask} is evaluated and, if the precondition is satisfied, \texttt{someTask} is executed like a synchronous operation. Thus, the task is transformed into a conditional method invocation.

### 3 System Model

The system model Hybster is based on is similar to the one used by PBFT [5, p. 26ff.]. Briefly, in a partially synchronous system with unreliable network, clients invoke operations of a stateful service that is replicated across several servers called replicas in order to ensure that the service does not return incorrect results even if a bounded number of these replicas behaves arbitrarily faulty. As the most notable difference to PBFT, however, Hybster also assumes that processes can be equipped with subsystems that are more reliable than other components such that these subsystems never fail in an arbitrary manner but only by crashing. Assuming such trusted subsystems, Hybster requires fewer replicas than PBFT to tolerate the same number of faults. A more detailed description of Hybster's system model is given in the following.

#### 3.1 Processes

Hybster is designed for a distributed system in which processes communicate exclusively via message passing over a shared network. Two types of processes are distinguished, \texttt{replicas} and
A set of replicas, also called replica group, provides a replicated realization of a service, with each replica hosting an instance of an implementation of that service. Clients invoke operations of the service by means of a local invocation handler instance encapsulating the communication with the replicas. The set of replicas in the system \( \{ r_0, r_1, \ldots, r_{n-1} \} \) is denoted by \( \mathcal{R} \) and the number of replicas by \( n \), that is, \( n \equiv |\mathcal{R}| \). Analogously, the set and numbers of clients is given by \( \mathcal{C} \equiv \{ c_0, c_1, \ldots, c_{n_{\text{clients}}-1} \} \) and \( n_{\text{clients}} \equiv |\mathcal{C}| \).

**Listing 1:** Specification of a deterministic stateful service implementation.

```plaintext
1 module Service,

3 state, \in S \leftarrow s_0

6 upon public call invoke(c \in \mathcal{C}, invno \in \mathbb{N}, svccmd \in \mathcal{O}) \rightarrow \mathcal{O}' do
7 \quad s', svcret := calculateResult(state, c, invno, svccmd)
8 \quad state := s'
9 \quad return svcret

11 declare internal call calculateResult(s \in S, c \in \mathcal{C}, invno \in \mathbb{N}, svccmd \in \mathcal{O}) \rightarrow S \times \mathcal{O}'

13 upon output call createStateSnapshot() \rightarrow S do
14 \quad return state

16 upon input call installStateSnapshot(s \in S) do
17 \quad state := s
```

The replicated service is assumed to be stateful and its implementation realized as a deterministic state machine. That is, starting from the same state and provided with the same command as input, the service implementation, if correct, always yields the same result independent of the instance and independent of the replica that executes it. Listing 1 shows the detailed specification of the service implementation. All possible states of the service’s deterministic state machine are represented in the set \( S \). Each instance of the type Service starts in the same initial state \( s_0 \in S \) that is assigned to the instance variable \( state_s \). The main interface of a service instance is its public invoke method. invoke expects a service command \( svccmd \) of the type \( \mathcal{O} \) that encapsulates the invoked service operation together with all required arguments. Moreover, invoke has to be provided with the identifier \( c \in \mathcal{C} \) of the client that issued the command and a sequence number \( invno \) assigned by the client to identify the particular invocation. To distinguish different types of sequence numbers, invocation numbers are of the type \( \mathbb{N} \). The method invoke does not implement the state transition itself. For that purpose, it relies on the internal method calculateResult that deterministically maps a specified current state and the invocation information comprising the client, its invocation number, and the service command to a new state and a return value of the type \( \mathcal{O}' \). Being dependent of the concrete service, the method calculateResult is not implemented but only declared. invoke calls calculateResult on the basis of the state referred to by the instance variable \( state_s \). When the call returns, it sets \( state_s \) to the new state and returns with the return value calculated by the command execution. Besides the main method for invoking operations of the realized service, the type Service offers methods to create and install snapshots of the service state.

The interface of the invocation handler that is used by clients to issue commands to the replicated service is depicted in Listing 2. The method startInvocation takes a service command
Listing 2: Specification of the invocation handler interface.

```plaintext
1 module InvocationHandler
2 declare input call startInvocation(svccmd ∈ O)
3 declare output task invocationCompleted(secret ∈ O')
```

of the type O as input. The respective return value of the command execution is eventually delivered by the output task invocationCompleted. Correct clients are only allowed to start a new invocation when a previous invocation was completed, that is, when no other invocation is in progress.

### 3.2 Time

The system is assumed to be partially synchronous [9]. More precisely, it is assumed that there exist upper bounds for the delivery of messages and how long it takes a process to carry out particular operations but that these upper bounds are not known a priori.

Listing 3: Specification of timers with individual clock speeds.

```plaintext
1 module Timer

3 upon init() do
4   timeout ∈ N ∪ {∅} := ∅

6 upon input call schedule(timeout ∈ N) do
7   timeout := timeout

9 upon internal task tick() do
10   with
11     timeout ≠ ∅ ∧ timeout > 0
12   do
13     timeout := timeout − 1

15 upon output task timerExpired() do
16   with
17     timeout ≠ ∅ ∧ timeout = 0
18   do
19     timeout := ∅

21 upon output call isScheduled() → Bool do
22   return timeout ≠ ∅

24 upon input call cancel() do
25   timeout := ∅
```

In this model, processes have access to local timers that advance at individual speeds. An upper bound for the relative speeds exists but is unknown. Listing 3 presents the specification of a type Timer that can signal when a configured amount of time that is relative to each Timer instance elapsed. For that purpose, each instance maintains a variable timeout that stores the remaining number of clock ticks until the timer expires or None if no timeout is currently configured. The input method schedule can be used to start the timer. It sets timeout to the specified number of clock ticks. The internal task tick regularly decrements the current value of timeout by one as long as timeout is greater than zero. Though, it depends on the relative...
speed of the timer instance in which interval the task is executed. Once the value of timeout reaches zero, the output task timerExpired signals that the previously configured number of ticks has elapsed. In addition to the basic functionality, the type Timer defines methods to determine if a timeout is currently configured and to stop the timer before timerExpired is triggered.

3.3 Communication

The network used by processes to communicate with each other by sending messages can be unreliable. It is allowed to reorder, duplicate, delay, and even drop messages. As a consequence, the assumed upper bound for message delivery only holds if a message is sent sufficiently often. Since the upper bound is not known a priori, it is not known in advance how often this is.

The specification of the network and its conditions can be found in Listing 4. Messages (Listing 4a) are modeled as tuples that contain at least one element identifying the message type. A message of the type MSG can be written as ⟨MSG,[…]⟩, where […] is a placeholder for a list of optional elements. From the network’s perspective, processes act as sinks that receive any kind of message (Listing 4b). The specification of the network itself (Listing 4c) is based on the one given by Castro [5, p. 28]. The state of a network comprises the set of all connected processes stored in nodes_n and the set wire_n that maintains a pair for all messages that are currently in transmission. The first element of such a pair is the message in question and the second is a set of processes supposed to receive the message. To request the network to transmit a message m, the input method send is provided. It takes m as argument together with a subset R of the connected processes and notes R in wire_n as intended recipients for m. Delivering messages is the responsibility of the internal task transmit. It removes a recipient r of an existing pair ⟨m,R⟩ in wire_n and invokes the receive method of r passing m as argument. If no recipient for a message m is left, the pair for m in wire_n can be cleaned up by the internal task discard. However, as long as the pair for a message m is contained in wire_n, it is possible that the set of recipients is altered by the task misbehave. Modeling the unreliable behavior of the network, misbehave can add processes to or remove processes from the current set of intended recipients for m. This includes adding new recipients to an empty set or removing all recipients. The only constraint at this point is that the task is fair, that it does not have any bias towards particular messages. The likelihood for being altered has to be the same for all pairs in wire_n to ensure that messages sent infinitely often are also delivered infinitely often and that messages sent finitely often are also delivered finitely often. Note that the network as specified here does not invent completely new messages, which includes that it does not corrupt sent messages. This is actually more restrictive than necessary. All messages in Hybster are authenticated (see below), which ensures their integrity at the protocol level even if the network could deliver corrupted messages.

3.4 Cryptography

Hybster relies on digital signatures and message authentication codes (MACs) to authenticate messages exchanged over the network. Moreover, cryptographic hash functions are used to reduce the amount of transmitted data.

The authenticity of messages is documented by message certificates (Listing 5a). Message certificates are messages themselves, they are tuples comprising a certificate type, a proof of their validity, and further elements if required. Usually, they are directly attached to the mes-
Listing 4: Specification of the communication system shared by all processes.

(a)

1 module Message
2 extends Tuple(MessageType, . . .)
3
4 declare output method type() → MessageType

(b)

1 module MessageSink
2 declare input call receive(m ∈ Message)

(c)

1 module Network
3
4 upon init() do
5 nodes_n ∈ P(MessageSink) := set of all connected processes
6 wire_n ∈ P(Message × P(MessageSink)) := {} 
7 upon input call send(m ∈ Message, R ∈ P(MessageSink))
8 asserts
9 if R ⊆ nodes_n
10 do
11 wire_n := wire_n \ {(m, R')} ∪ {(m, R' ∪ R)}
12 else
13 wire_n := wire_n ∪ {(m, R)}

14 upon internal task transmit(m ∈ Message, r ∈ MessageSink)
15 with
16 ∃(m, R) ∈ wire_n : r ∈ R
17 do
18 wire_n := wire_n \ {(m, R)} ∪ {(m, R \ {r})}
19 output r.receive(m)

20 upon internal task discard(m ∈ Message)
21 with
22 ∃(m, ∅) ∈ wire_n
23 do
24 wire_n := wire_n \ {(m, ∅)}

25 upon internal task misbehave(m ∈ Message, R ∈ P(MessageSink), R' ∈ P(MessageSink))
26 with
27 ∃(m, R) ∈ wire_n
28 R' ⊆ nodes_n
29 do
30 wire_n := wire_n \ {(m, R)} ∪ {(m, R')}

sage they certify forming a certified message (Listing 5b). The notation for such a message is ⟨. . .⟩[certificate], with [certificate] being a placeholder for an identifier of the employed certification method. ⟨. . .⟩_σ_x denotes a message signed by a process x. If a message is certified by a MAC, it is written as ⟨. . .⟩_µ_x,y,z, where the underlying secret key is shared among the processes x, y, and z. If a process x issues an array of MACs, a so-called authenticator [7] for the processes y and z, the message is annotated with ⟨. . .⟩_α_x,y,z.
Listing 5: Specification of certified messages.

(a)

module MessageCertificate
extends Message

declare output method proof() → Proof

(b)

module CertifiedMessage
extends Message

declare output method certificate() → MessageCertificate

Listings 6a and 6b depict the specifications for providers of signatures and MACs, respectively. An instance of the type SignatureProvider is initialized with a public/private key pair. It issues digital signatures on the basis of this pair in the method createSignature and verifies signatures in verifySignatures using the public key connected to the signature in question. Message authentication codes are issued by MacProvider instances. MACs are created and verified in the methods createMac and verifyMac using the shared key that is configured during the initialization of the instance.

The implementation of a cryptographic hash function is provided by the type DigestProvider as shown in Listing 6c. This type offers a global method digest that returns the hash calculated for a given argument in form of a message similar to the certificates issued by certification providers described above. The hash function is required to be collision resistant such that \( \forall m, m': \text{digest}(m) = \text{digest}(m') \leftrightarrow m = m' \) holds with almost absolute certainty during the period in which the message \( m \) is relevant and not outdated from perspective of the protocol.

3.5 Faults

Constraints. A process, be it a replica or a client, can fail at any point in time. The behavior of a failed process is undetermined, it can stop the processing of messages completely or it can start to act maliciously by sending messages that are not correct with regard to the protocol specification. Thus, processes can fail arbitrarily, that is, they can be Byzantine. However, this does not apply to all components of replicas. Part of each replica is a trusted subsystem that is assumed to fail only by crashing. That is, independent of the rest of the process, it is assumed that a trusted subsystem either behaves according to the specification or does not carry out any operation, it is assumed that a trusted subsystem never yields a result that is incorrect. Nonetheless, a process is still regarded as either correct or faulty. If some of the components of a process are faulty in any way, the whole process is deemed as faulty.

Furthermore, it is assumed that the cryptographic assumptions hold under all circumstances. Even if an adversary controlled all faulty processes in the system, it is assumed that the collaborating processes would not be able to impersonate correct processes or find collisions for message hashes. Therefore, adversaries are regarded as computationally restricted.

Given a point \( t \) in the time \( T \) of some execution of the system, \( \mathcal{R}_f(t) \) denotes the set of all replicas that are faulty at \( t \) and \( \mathcal{C}_f(t) \) the set of all faulty clients. The recovery of processes is not considered. Therefore, it holds: \( \forall t, t' \in T : t \leq t' \rightarrow (\mathcal{R}_f(t) \cup \mathcal{C}_f(t)) \subseteq (\mathcal{R}_f(t') \cup \mathcal{C}_f(t')) \). Further, if a replica or a client is said to be faulty without a reference to time, there is some
Listing 6: Specification of providers for message certificates and cryptographic hashes.

(a)

1 module SignatureProvider,

3 upon init(pubkey ∈ PublicKey, privkey ∈ PrivateKey) do
4 pubkey, := pubkey
5 privkey, := privkey

7 upon output call createSignature(m) → MessageCertificate do
8 p := signature for m using privkey, return (Signature, pubkey, p)

11 upon output call verifySignature((Signature, pubkey, p), m) → Bool do
12 return use pubkey to verify that p is a valid signature for m

(b)

1 module MacProvider,

3 upon init(key ∈ SharedKey) do
4 key, := key

6 upon output call createMac(m) → MessageCertificate do
7 p := calcMac(m)
8 return (Mac, p)

10 upon output call verifyMac((Mac, p), m) → Bool do
11 return p = calcMac(m)

13 upon internal call calcMac(m) → Mac do
14 return MAC for m using key

(c)

1 module DigestProvider

3 upon global call digest(m) → Message
4 d := calculate digest of m
5 return (Digest, d)

time in the execution in question in which it does not behave according to the specification. As a consequence, the set of faulty replicas is given by \( \mathcal{R}_f \equiv \bigcup_{t \in T} \mathcal{R}_f(t) \) and the set of faulty clients by \( \mathcal{C}_f \equiv \bigcup_{t \in T} \mathcal{C}_f(t) \). It is assumed that the number of faulty replicas does not exceed a particular \( f \), thus it is assumed that \( |\mathcal{R}_f| \leq f < n \) is always true. Opposed to that, there is no required upper bound for the number of faulty clients, that is, \( |\mathcal{C}_f| \leq n_{clients} \). Complementing the sets of faulty processes, the sets for correct replicas and clients are defined as \( \mathcal{R}_c \equiv \mathcal{R} \setminus \mathcal{R}_f \) and \( \mathcal{C}_c \equiv \mathcal{C} \setminus \mathcal{C}_f \), respectively.

Quorums. Since processes can fail silently or omit to send messages and since the assumed upper bounds for message delivery and processing times are not known, a correct process cannot know if an expected message from another process \( p \) has not arrived yet because \( p \) is faulty or because the network or \( p \) are slower than currently assumed. Hence, in situations where messages
are not received in time, processes cannot determine if another process \( p \) is faulty or not. Even if \( p \) provides a correct message in time, it can be Byzantine or fail at a later point in time. As a consequence, state-machine replication protocols do not rely on single replicas to acknowledge protocol information or states but on sufficiently large subsets of the replica group, so-called *quorums* [7]. Formally, a quorum is defined as a set \( Q \) such that \( Q \subseteq R \land |Q| \geq qs \), with \( qs \) being a minimum quorum size that depends on the properties the particular quorum shall provide. The most important quorums are called *intersecting quorums*. If not stated otherwise, these quorums have the fixed minimum size \( q \) and fulfill the properties that (1) there is at least one, possibly faulty, replica in the intersection of every two quorums \( 2q > n \) and (2) the number of correct replicas in the system suffices to form a quorum \( n \geq q + f \). An important implication of these properties is that each intersecting quorum contains at least one correct replica \( q > f \). Given a number of replica faults \( f \) that has to be tolerated, these properties lead to a minimum configuration of \( n = 2f + 1 \) required replicas with a minimum size for intersecting quorums of \( q = f + 1 \). Another type of quorums are *acknowledging quorums*, also called weak quorums. These quorums have the property that at least one correct replica is contained. Thus, their minimum size is always \( f + 1 \), independent of the number of replicas \( n \). Further, in the minimum configuration they have the same size as intersecting quorums.

4 Specification of Hybster

This section describes the properties Hybster provides and presents a formal specification of the protocol. An informal step-by-step description will be part of a future version of this document.

4.1 System Properties

Hybster provides roughly the same properties as PBFT [5], which is a form of linearizability [10] that takes the arbitrary behavior of Byzantine clients into account. From the perspective of each correct client, the replicated system appears to execute all invocations sequentially in an order that respects the real-time relation between the invocations.

The specification of the properties Hybster satisfies is presented in Listing 7 and is based on the one given by Castro [5, p. 30]. It uses the service and invocation handler specifications introduced in Section 3.1.

Initialization. First, a couple of system-wide variables are initialized \((\text{init}())\). \( \text{svc} \) takes an instance of the service implementation. The set of issued but yet unprocessed invocations is maintained in \( \text{pending} \) and all undelivered responses in \( \text{responses} \). Invocations are triples of a client \( (C) \), an invocation number \( (iN) \), and a service command \( (O) \), thus \( I \equiv C \times iN \times O \). Invocation numbers are individual to each client and correct clients assign these numbers only once and monotonically increasing. Therefore, invocations can be identified by the pair of a client and an invocation number \( (C \times iN) \). A response to an invocation is formed by its identifier and the value returned by its execution \( (O') \), that is, \( Y \equiv C \times iN \times O' \). Moreover, the system-wide variable \( \text{last}\_\text{inv} \) references a vector that stores for each client the invocation number of the last executed invocation. All entries of this vector are initialized with zero and the first valid invocation number for each client is one.
Listing 7: Specification of linearizability with Byzantine clients.

1 module ByzantineLinearizability,
2 extends InvocationHandler as c

4 upon init(), do
5 \text{svc}_c := \text{Service}()
6 pending_c \in \mathcal{P}(\mathcal{X}) := \{\}
7 responses_c \in \mathcal{P}(\mathcal{Y}) := \{\}
8 last_invsc : \mathcal{C} \rightarrow \mathbb{N}, \text{for all } c \in \mathcal{C} \text{ do } last_invsc(c) := 0

10 upon init(), do
11 invno_c \in \mathbb{N} := 0
12 svccmd_c \in \mathcal{O} \cup \{\emptyset\} := \emptyset
13 isfaulty_c \in \text{Bool} := \text{False}

15 upon input call \text{startInvocation}({\text{svccmd}_c} \in \mathcal{O})_c
16 asserts
17 \text{svccmd}_c = \emptyset
18 do
19 invno_c := invno_c + 1
20 svccmd_c := svccmd
21 registerInvocation((c, invno_c, svccmd))

23 upon internal call \text{registerInvocation}(\text{inv}' = (c \in \mathcal{C}, . . .))_c do
24 pending_c := pending_c \setminus \{\text{inv} \in \text{pending}_c | \text{inv} = (c, . . .) \} \cup \{\text{inv}'\}

26 upon internal task \text{processInvocation}(\text{inv} = (c \in \mathcal{C}, \text{invno} \in \mathbb{N}, \text{svccmd} \in \mathcal{O}))_c
27 with
28 \text{inv} \in \text{pending}_c, \text{invno} > \text{last_invsc}_c(c) \land \text{preservesFairness(\text{inv})}
29 do
30 pending_c := pending_c \setminus \{\text{inv}\}
31 secret := \text{invoke} \text{svc}_c, \text{invoke}(c, \text{invno}, \text{svccmd})
32 responses_c := responses_c \setminus \{\text{resp} \in \text{responses}_c | \text{resp} = (c, . . .) \} \cup \{c, \text{invno}, \text{secret}\}
33 last_invsc_c(c) := \text{invno}

35 declare internal call \text{preservesFairness}(\text{inv} = (c \in \mathcal{C}, \text{invno} \in \mathbb{N}, \text{svccmd} \in \mathcal{O}))_c

37 upon output task \text{invocationCompleted}({\text{secret} \in \mathcal{O'}})_c
38 with
39 isfaulty_c \lor \text{resp} = (c, \text{invno}_c, \text{secret}) \in \text{responses}_c
40 do
41 responses_c := responses_c \setminus \{\text{resp}\}
42 svccmd_c := \emptyset

44 upon internal task \text{clientFailureOccured}().c
45 with
46 \neg isfaulty_c
47 do
48 isfaulty_c := True

50 upon internal task \text{injectFaultyInvocation}(c \in \mathcal{C}, \text{invno} \in \mathbb{N}, \text{svccmd} \in \mathcal{O})_c
51 with
52 isfaulty_c
53 do
54 registerInvocation((c, \text{invno}, \text{svccmd}))
Invocation. Invocation handler instances are dedicated to single clients and used by them to invoke operations of the replicated service. A handler instance maintains three variables (see \texttt{init()}, \texttt{invno}_c, \texttt{svccmd}_c), the invocation number of the last invocation \texttt{invno}_c, \texttt{svccmd}_c that stores the service command of a currently running but not yet completed invocation or $\emptyset$ if no invocation is in progress, and the flag \texttt{isfaulty}_c that marks faulty clients. Correct clients start only one invocation at a time and wait for its completion before they start a new one. Hence, the input method \texttt{startInvocation} expects that \texttt{svccmd}_c is $\emptyset$ when it is called. Subsequently, it assigns the next invocation number to the passed service command, sets \texttt{svccmd}_c to mark the invocation as in progress, and submits the new invocation by calling \texttt{registerInvocation}. This system-wide method adds invocations to the set of pending invocations while ensuring that each client has at most one outstanding invocation in that set.

Processing. Pending invocations are delivered to the service instance by the task \texttt{process-Invocation}, one invocation at a time. It considers only invocations with numbers higher than the numbers stored in \texttt{last_inv}s, that is, with numbers higher than the one of the last invocation that was executed for the respective client. Being always the case for correct clients, this prevents faulty clients from injecting invocations in a non-increasing order. Further, \texttt{processInvocation} makes use of the method \texttt{preservesFairness} to select the invocation to be executed next. This auxiliary method is only declared and documents the requirement that the selection mechanism is unbiased and does not give particular clients an advantage. Once an invocation has been chosen, it is removed from the pending set and handed over to the service implementation. The value returned by the service is used to create a response. Before this response is added to the set of undelivered responses, older responses for the client are removed if necessary. A faulty client could have sent a new invocation without collecting the response for a previous one. Finally, \texttt{last_inv}s is updated to mark the current invocation as executed.

Completion. Adhering to the interface specified by the type \texttt{InvocationHandler}, invocations are delivered to clients through the output task \texttt{invocationCompleted}. Provided that a client $c$ is not marked as faulty, this task removes a response belonging to $c$ from the set of undelivered responses and marks the invocation as completed by setting \texttt{svccmd}_c to $\texttt{None}$ before it delivers the return value.

Faulty Clients. A client $c$ can fail at any point in time. This is modeled by the task \texttt{clientFailureOccured}. If executed, it simply sets the flag \texttt{isfaulty}_c to $\texttt{True}$. From that time on, the client $c$ can circumvent the regular \texttt{startInvocation} method and can issue arbitrary invocation as realized by the task \texttt{injectFaultyInvocation}. However, it can still only do so for invocations that are attributed to $c$ itself. As stated in Section 3.5, the system model assumes that even arbitrarily behaving clients are not able to impersonate correct clients. In addition to the injection of arbitrary invocation, a faulty client can locally deliver any return value as reflected in the task \texttt{invocationCompleted}.

4.2 Auxiliary Modules

The specifications of Hybster’s main protocols make use of two mechanisms that help to cope with the partially synchronous environment and the unreliable network as assumed by the system model (Section 3): adaptive timeouts and retransmitting connections. The respective modules realizing these mechanisms are presented in the following.
4.2.1 Adaptive Timeouts

The time model Hybster’s specification is based on assumes upper bounds for message delivery and processing times, upper bounds, however, that are unknown (cf. Section 3.2). As a consequence, it cannot be known how long certain protocol actions need to finish. Nonetheless, it is known that they do not require an infinite amount of time as long as their completion does not depend on a faulty process. Therefore, provided that some protocol action can be retried infinitely often, it is possible to approach the upper bound by increasing the value for the timeout that determines when the action is regarded as failed.

Listing 8: Specification of timeouts adaptive to unknown upper bounds for delays.

```plaintext
1 module AdaptiveTimeout,
3 upon init(procs ∈ Set) do
4 procs_s := procs
5 timer_s := Timer()
6 delay_delta_s ∈ N := increment if timeout expired
7 delay_s ∈ N := delay_delta_s
8 expired_s ∈ Bool := False
10 upon input call start(x)
11 asserts
12 x ∈ procs_s, // x could be used in more sophisticated implementations
13 do
14 adapt()
15 output timer_s.schedule(delay_s)
17 upon input call adapt() do
18 if expired_s then
19 delay_s := delay_s + delay_delta_s
20 expired_s := False
22 upon input call stop() do
23 output timer_s.cancel()
24 expired_s := False
26 upon internal task timerExpired() with
28 input timer_s.timerExpired()
29 do
30 expired_s := True
32 upon output call isExpired() → Bool do
33 return expired_s
35 upon output call isRunning() → Bool do
36 return input timer_s.isScheduled()
```

This mechanism is realized in the module AdaptiveTimeout shown in Listing 8. It is intended that one instance of this module is used for one particular protocol action that has to be carried out by one process from a group of processes. For example, if a replica is supposed to provide a particular message, an instance of this module can be used to signal when this step in the protocol is considered as failed. If this is the case, the step is repeated, but this time, a different replica is chosen to provide the message and it is given more time to do so. Eventually, this
step will complete successfully, at the latest when a correct replica is selected and the currently configured timeout exceeded the upper bound for providing the expected message.

Although the implementation of AdaptiveTimeout is tailored to the time model stated above, please note that AdaptiveTimeout in fact abstracts from the concrete model. Currently, it increases the timeout value linearly. If a time model as in PBFT were used in which a delay function \( \text{delay}(t) \) does not grow faster than the time \( t \) [5], an exponential increase could compensate for that. More sophisticated strategies would be required if the time bounds were only assumed to hold long enough such that the system is able to make progress [9]. Still, higher-level protocols could rely on the abstraction provided by AdaptiveTimeout.

The module AdaptiveTimeout in detail: Each instance maintains a reference to a timer instance, the current timeout value stored in \( \text{delay}_s \) and initialized with a configured constant \( \text{delay}_\text{delta}_s \), and a flag \( \text{expired}_s \) set when the timeout expired. The instance also gets the set of processes that are within the considered group. This set is actually not required by the presented implementation and is only provided to allow for other strategies that adjust the timeout value for processes individually instead of globally for all processes. The timeout is started through the input method \( \text{start} \). It takes the process as argument that is currently expected to carry out the monitored action. Again, the presented implementation does not make use of this information. Upon being invoked, the method \( \text{start} \) configures the timer to signal after time \( \text{delay}_s \). Prior to that, it is checked if the current timeout value \( \text{delay}_s \) needs to be adjusted. The strategy for that is realized by the input method \( \text{adapt} \). If the timer was marked as expired, it adds \( \text{delay}_\text{delta}_s \) to \( \text{delay}_s \) and resets the expired flag. If the timed protocol action completed or if the timeout is not needed anymore, the input method \( \text{stop} \) can be invoked canceling a currently running timer. Otherwise, the timer will eventually expire, monitored with the internal \( \text{timerExpired} \) task. This sets the expired flag which can be queried by the output method \( \text{isExpired} \). Additionally, \( \text{isRunning} \) returns if the timer is currently scheduled or not.

### 4.2.2 Connections

Even if a process expected to provide some message were correct and even if other processes waited actually long enough to receive this message, the message might not come in time since the network could have dropped it. The upper bounds for message delivery only hold if the message is sent sufficiently, virtually infinitely often (see Section 3.3).

Retransmitting messages is the responsibility of the Connection module presented in Listing 9. It sends registered messages over and over again until they are explicitly removed from the connection by a higher-level protocol. An instance of the Connection module expects a set of recipients when constructed. It further has a reference to the network and keeps with \( \text{out}_s \) a set of all messages that are currently in transmission. Messages are handed over to the connection via the \( \text{send} \) input method which adds the message to \( \text{out}_s \). The internal task \( \text{retransmit} \) is enabled as long as some message is contained in \( \text{out}_s \). When executed, it picks one message and (re-)sends it to the configured recipients. Nevertheless, the selected message remains in the set \( \text{out}_s \) and could be chosen again in a later execution of the task. If multiple messages are contained in \( \text{out}_s \), the selection mechanism is supposed to be fair, on average all messages are retransmitted at the same rate. The module offers multiple variants of the method \( \text{remove} \) used to deregister messages from the connection. \( \text{remove} \) can be invoked with a single message, a set of messages, or a predicate that determines which messages are to be removed from \( \text{out}_s \).

1 module Connection,

3 upon init(recipients ∈ P(MessageSink)) do
4    recipients ∈ Network := reference to the network
5    recipients, := recipients
6    out, ∈ P(Message) := {}

8 upon input call send(m ∈ Message) do
9    out, := out, ∪ {m}

11 upon internal task retransmit(m ∈ Message) with
12    m ∈ out,
13    do
14        output net,, send(m, recipients)

17 upon input call remove(m ∈ Message) do
18    out, := out, \ {m}

20 upon input call remove(M ∈ P(Message)) do
21    out, := out, \ M

23 upon input call remove(P: Message → Bool) do
24    out, := out, \ {m ∈ out, | P(m)}
4.3 TrInX

Listing 10: Specification of the trusted subsystem TrInX.

 module TrInX

 upon init(id ∈ TssID, ctrtypes ∈ Set, key ∈ SharedKey) do
  insts := id
  ctrtypes := ctrtypes
  macprov := MacProvider(key)
  ctrs := ctrtypes → N, for all tc ∈ ctrtypes do ctrs(tc) := 0

 upon public call createContinuingCertificate(tc, tv' ∈ N, m) → MessageCertificate
  asserts
  tc ∈ ctrtypes ∧ tv' ≥ ctrs(tc)
  do
  p := macprov.createMac(insts∥tv'||tv∥m)
  ctrs(tc) := tv'
  return ⟨TCtr, insts, tc, tv', tv, p⟩

 upon public call createIndependentCertificate(tc, tv' ∈ N, m) → MessageCertificate
  asserts
  tc ∈ ctrtypes ∧ tv' > ctrs(tc)
  do
  p := macprov.createMac(insts∥tv'∥tv∥m)
  ctrs(tc) := tv'
  return ⟨TCtr, insts, tc, tv', −, p⟩

 upon output call verifyCertificate(TCtr, insts, tc, tv', tv, p, m) → Bool do
  return macprov.verifyMac(insts∥tv'||tv∥m)

 upon input call forwardCounter(tc, tv' ∈ N)
  asserts
  tc ∈ ctrtypes ∧ tv' ≥ ctrs(tc)
  do
  ctrs(tc) := tv'

 http://publikationsserver.tu-braunschweig.de/get/64440
4.4 Processes

4.4.1 Clients

Listing 11: Specification of invocation handlers dedicated to a client.

```java
1 module ByzInvocationHandler_c
2 extends InvocationHandler
3 extends MessageSink
4
5 upon init(pubkey ∈ PublicKey, privkey ∈ PrivateKey, pubkeys: R → PublicKey) do
6 reppconns: R → Connection, for all r ∈ R do reppconns(r) := Connection({r})
7 repgrpconn := SignatureProvider(pubkey, privkey)
8 pubkeys_c := pubkeys
9 imc := InvocationHandler_c::Invocation()
10
11 upon input call startInvocation(svccmd ∈ O) do
12 output imc.startInvocation(svccmd)
13
14 upon output task invocationCompleted(svcret ∈ O') with
15 input imc.invocationCompleted(svcret)
16
17 upon input call receive(m ∈ Message) do
18 output imc.receive(m)
19
20 upon internal call hasValidCertificate(m = ⟨...⟩sr) → Bool do
21 p := m.certificate().proof()
22 return input sigprov_c.verifySignature((SIGNATURE, pubkeys_c(r), p), m)
```

4.4.2 Replicas

Listing 12: Specification of the current view status used by protocol modules.

```java
1 module ViewStatus_s
2
3 upon init(replica ∈ R) do
4 replica_s ∈ R := replica
5 vstab_s ∈ ℤ_N := 0
6 vcur_s ∈ ℤ_N := vstab_s
7
8 upon global call leader(v ∈ ℤ_N) → R do
9 return rv mod n
10
11 upon input call leaveViewFor(v' ∈ ℤ_N) asserts
12 vcur_s < v'
13 do
14 vcur_s := v'
15
16 upon input call enterView(v' ∈ ℤ_N) asserts
17 vstab_s < v' ≤ vcur_s
18 do
19 vstab_s := v'
20
21 upon output call latestLeader() → R do
22 return leader(vstab_s)
```
upon output call isInStableView() \rightarrow \text{Bool} do
return \text{_cur}_s = \text{_stab}_s

upon output call isStableLeader() \rightarrow \text{Bool} do
return \text{isInStableView()} \land \text{replicas} = \text{lastestLeader()}

upon output call isStableFollower() \rightarrow \text{Bool} do
return \text{isInStableView()} \land \text{replicas} \neq \text{lastestLeader()}

upon output call stableView() \rightarrow \text{N} do
return \text{v_stab}

upon output call currentView() \rightarrow \text{N} do
return \text{v_cur}


1 module Replica,
2 extends MessageSink

4 def TC := \{M, O, N\}

6 upon init(pubkey \in \text{PublicKey}, privkey \in \text{PrivateKey},
\hspace{1cm} tsskey \in \text{SharedKey}, pubkeys : \mathcal{R} \cup \mathcal{C} \rightarrow \text{PublicKey}, tssinsts : \mathcal{R} \rightarrow \text{TssID} \}) do
7 repconn_r : \mathcal{R} \setminus \{r\} \rightarrow \text{Connection}, for all r \in \mathcal{R} \setminus \{r\} do repconn_r(r) := \text{Connection}(\{r\})
8 repgrconn := \text{Connection}(\mathcal{R} \setminus \{r\})
9 tss_r := \text{TrInX}(tssinsts(r), TC, tsskey)
10 tssinsts_r := tssinsts
11 sigprov_r := \text{SignatureProvider}(pubkey, privkey)
12 pubkeys_r := pubkeys
13 cm_r := Replica_r::Client
14 om_r := Replica_r::Ordering
15 km_r := Replica_r::Checkpointing
16 vm_r := Replica_r::ViewChange
17 em_r := Replica_r::Execution
18 sm_r := Replica_r::StateTransfer

20 upon input call receive(m \in \text{Message}) do
21 // dispatch message according to message type
22 output (cm_r, om_r, km_r, vm_r, em_r, sm_r).receive(m)
24 upon internal call hasValidCertificate(m = \{\ldots\}_x \rightarrow \text{Bool} do
25 p := m.certificate().proof()
26 return input sigprov_r.verifySignature((\text{Signature, pubkeys}_r(x), p), m)
28 upon internal call hasValidCertificate(m = \{\ldots\}_r(r', tc, tv, tv') \rightarrow \text{Bool} do
29 p := m.certificate().proof()
30 return input tss_r.verifyCertificate((TC, tssinsts_r(r'), tc, tv, tv, p), m)
4.5 Invocation Protocol

4.5.1 Client Side

Listing 14: Specification of the client role of the invocation protocol.

```plaintext
1 module InvocationHandler_c::Invocation

2 upon internal call leader(v ∈ ℤN) → R do
3    return r_v mod n

4 upon init() do
5    v_c ∈ ℤN := 0
6    invno_c ∈ ℤN := 0
7    req_c ∈ Message∪{∅} := ∅
8    reps_c ∈ P(Message) := {} 
9    rq_timeout_c := AdaptiveTimeout(R) 
10   retrans_c ∈ Bool := False 

11 upon input call startInvocation(svccmd ∈ O) 
12    asserts
13    req_c = ∅ 
14    do
15       invno_c := invno_c + 1 
16       req_c := (REQUEST, c, invno_c, svccmd)_c 
17       sigprov_c.createSignature(req_c) 
18       output repconns,(leader(v_c)).send(req_c) 
19       rq_timeout_c.start(leader(v_c))

20 upon internal task retransmit() 
21   with
22       req_c ≠ ∅ ∧ ¬retrans_c ∧ rq_timeout_c.isExpired() 
23       do
24          rq_timeout_c.adapt() 
25          output repgrpconn,(leader(v_c)).send(req_c) 
26          retrans_c := True

27 upon global call isCorrectReply(rp, r ∈ R, v ∈ ℤN, c′ ∈ C, invno ∈ ℤN, svcret ∈ O) do
28    return rp = (REPLY, r, v, c′, invno, svcret)_c ∧ hasValidCertificate(rp) 

29 upon input call receive(rp′ = (REPLY, r, v′, ·, ·, secret′)...) do
30    if req_c ≠ ∅ ∧ isCorrectReply(rp′, ·, ·, invno_c) then
31       if ∃v ≠ v′∧ secret ≠ secret then
32          reps_c := reps_c ∪ {rp′}
33       else if secret′ ≠ secret then
34          // r is faulty
35       else if v′ > v then
36          reps_c := reps_c ∪ {rp′} \ {rp}

37 upon global call isCorrectReplyCertificate(Y ⊆ Message, c′ ∈ C, invno ∈ ℤN, svcret ∈ O) do
38    return |Y| > f
39       ∧ (∀rp ∈ Y ∃r ∈ R: isCorrectReply(rp, r, ·, ·, c′, invno, svcret)
40         ∧ (∀rp′ ∈ Y: rp′ = (REPLY, r, ..) → rp′ = rp))

41 upon output task invocationCompleted(svcret ∈ O) 
42    with
43       ∃Y ⊆ reps_c: isCorrectReplyCertificate(Y, c, invno_c, secret)
44    do
```

http://publikationsserver.tu-braunschweig.de/get/64440
4.5.2 Replica Side

Listing 15: Specification of the replica role of the invocation protocol.

```plaintext
module ReplicaStr::Client

def global ReturnEntry \(\equiv \Pi \times (O' \cup \{\varnothing\})\)

upon global call invocationNumber((invno, svcret) \in ReturnEntry) \rightarrow \Pi
return invno

upon init() do
  cstatus := ViewStatus(r)
  reqs := P(Message) := \{\}
  \(rq\_timeout_s, c) := AdaptiveTimeout\)
  \(last\_retvals_s, c) := \{0, \varnothing\)
  cliconns, (c) := Connection({c})
  for all \(c \in \mathbb{C}\) do
    \(rq\_timeout_s, c) := AdaptiveTimeout(R)
    \(last\_retvals_s, c) := \{0, \varnothing\)
    cliconns, (c) := Connection({c})

upon global call isCorrectRequest((rq, c) \in C, invno \in \Pi, svccmd \in O) do
return \(rq = (\text{REQUEST}, c, invno, svccmd)_{e_s} \land \text{hasValidCertificate}(rq)\)

upon call receive((rq', \{\text{REQUEST}, c, invno', svccmd\}) \ldots) do
  if isRelevantRequest((rq') \land isCorrectRequest((rq', \ldots)) then
    storeRequest((rq'))
    if cstatus, isStableFollower() then
      forwardRequest((rq'))
      startRequestTimeout(c)

upon internal call isRelevantRequest((rq' = (\text{REQUEST}, c, invno', \ldots)) do
return \(\neg isRequestExecuted((rq')) \land (\forall r \in reqs, \exists invno \geq invno', r) = (\text{REQUEST}, c, invno, \ldots)\)

upon internal call storeRequest((rq' = (\text{REQUEST}, c, \ldots)) do
  reqs := reqs \{\{r \in reqs, r = (\text{REQUEST}, c, \ldots)\}\}

upon internal call forwardRequest((rq = (\text{REQUEST}, c, \ldots)) do
  output repconns, (contact\(_s\). remove\{\{m | m = (\text{REQUEST}, c, \ldots)\}\}

upon output call pendingRequests() \rightarrow P(Message) do
return reqs,

upon input call requestExecuted((rq' = (\text{REQUEST}, c, invno', svccmd), \ldots, svcret \in O') do
```

http://publikationsserver.tu-braunschweig.de/get/64440
\[\text{reqs} := \text{reqs} \setminus \{\text{rq} | \text{rq} = (\text{REQUEST}, c, \text{invo}, \cdot) \land \text{invo} \leq \text{invo}'\}\]

\[\text{last}_r\text{etvals}_r(c) := (\text{invo}', \text{secret})\]

\[\text{sendReply}(c, \text{invo}', \text{secret})\]

\[\text{if} \not\exists (\text{REQUEST}, c, \cdot \ldots) \in \text{reqs} \land \text{rq}_r\text{timeouts}(c).\text{isRunning()} \text{ then}\]

\[\text{rq}_r\text{timeouts}(c).\text{stop()}\]

\[\text{upon output call isRequestExecuted}(\text{rq} = (\text{REQUEST}, c, \text{invo}, \cdot) \ldots) \text{ do}\]

\[\text{return invno} \leq \text{invocationNumber}(\text{last}_r\text{etvals}_r(c))\]

\[\text{upon internal call sendReply}(c \in C, \text{invo} \in \mathbb{N}, \text{secret} \in O') \text{ do}\]

\[\text{rp} := (\text{REPLY}, r, \text{cstatus}_r, \text{stableView}(), c, \text{invo}, \text{secret})_s.\]

\[\text{sigprov}_r.\text{createSignature}(\text{rp})\]

\[\text{output cliconns}_r(c).\text{remove}\{m | m = (\text{REPLY}, r, \cdot, c, \cdot \ldots)\}\]

\[\text{output cliconns}_r(c).\text{send}(\text{rp})\]

\[\text{upon output call createReturnValueMapSnapshot()} \rightarrow (C \rightarrow \text{ReturnEntry}) \text{ do}\]

\[\text{return last}_r\text{etvals}_r,\]

\[\text{upon input call installReturnValueMap}(\text{retvals} \in C \rightarrow \text{ReturnEntry}) \text{ do}\]

\[\text{last}_r\text{etvals}_r, := \text{retvals}\]

\[\text{for all } \text{rq} = (\text{REQUEST}, c, \cdot \ldots) \in \text{reqs} \text{ do}\]

\[\text{if isRequestExecuted}(\text{rq}) \text{ then}\]

\[\text{reqs} := \text{reqs} \setminus \{\text{rq}\}\]

\[\text{if rq}_r\text{timeouts}(c).\text{isRunning()} \text{ then}\]

\[\text{rq}_r\text{timeouts}(c).\text{stop()}\]

\[\text{upon internal call startRequestTimeout}(c \in C) \text{ do}\]

\[\text{if rq}_r\text{timeouts}(c).\text{isRunning()} \text{ then}\]

\[\text{rq}_r\text{timeouts}(c).\text{stop()}\]

\[\text{upon output call suspectsLeader()} \text{ do}\]

\[\text{return } \exists \text{to} \in \text{rq}_r\text{timeouts}_r: \text{to.isExpired()}\]

\[\text{upon input call leaveViewFor}(v' \in \mathbb{V}) \text{ do}\]

\[\text{cstatus}_r.\text{leaveViewFor}(v')\]

\[\text{upon input call enterView}(v' \in \mathbb{V}) \text{ do}\]

\[\text{cstatus}_r.\text{enterView}(v')\]

\[\text{if cstatus}_r.\text{isInStableView} \text{ then}\]

\[\text{output repconns}_r(\text{contact}_r).\text{remove}\{\text{rq} | \text{rq} = (\text{REQUEST}, \ldots)\}\}

\[\text{contact}_r := \text{cstatus}_r.\text{latestLeader}()\]

\[\text{for all } \text{rq} = (\text{REQUEST}, c, \cdot \ldots) \in \text{reqs} \text{ do}\]

\[\text{if cstatus}_r.\text{isStableFollower}() \text{ then}\]

\[\text{forwardRequest}(\text{rq})\]

\[\text{startRequestTimeout}(c)\]

\[\text{http://publikationsserver.tu-braunschweig.de/get/64440}\]
4.6 Ordering Protocol

Listing 16: Specification of the ordering protocol.

```ocaml
module Replica_r = Ordering

uses vm_r ∈ Replica_r::ViewChange
uses cm_r ∈ Replica_r::Client
uses em_r ∈ Replica_r::Execution

// An abort state comprises a prepared set \( P \) and
// the value of the trusted counter \( D \) at the time of the abort.
def global AbortState ≜ \( (P ∈ \mathcal{P}(\text{Message})) \times \mathbb{N} \)

let order_wnd ∈ \( \mathbb{N} \)

upon global call orderWindowSize() do
  return order_wnd

upon init() do
  ostatus_r := ViewStatus(r)
  omgs_r ∈ \( \mathcal{P}(\text{Message}) \) = \{ \}
  o_act_r ∈ \( \mathbb{N} \) := 0
  o_base_r ∈ \( \mathbb{N} \) := 0
  o_comm_r ∈ \( \mathbb{N} \) := o_base_r
  o_prep_r ∈ \( \mathbb{N} \) := o_comm_r
  o_max_r def := o_base_r + order_wnd

upon internal task proposalReady(rq = \( \langle \text{REQUEST}, c, invno, svccmd \rangle \)_σ) with
  o′ = o_prep_r + 1
  ostatus_r.isStableLeader() ∧ o′ ≤ o_max_r
  v = ostatus_r.nextStableView()
  rq ∈ input cm_r.pendingRequests() ∨ (\( \exists o : \langle \text{Prepare}, r, v, o, rq \rangle \) ∈ omgs_r)
  o′′ = \( o_{\text{act}} \) := o′
  o_prep_r := o′

upon internal call sendPrepare(v ∈ \( \mathbb{N} \), o ∈ \( \mathbb{N} \), rq ∈ \( \text{Message} \)) do
  pr := \( \langle \text{Prepare}, r, v, o, rq \rangle |\subseteq \langle (l, O, v | o, pr) \rangle \)
  invoke tss_r.createIndependentCertificate(O, v | o, pr)
  omgs_r := omgs_r ∪ \{pr\}
  output repgrpconn_r.send(pr)

upon global call isCorrectPrepare(pr, l ∈ \( \mathcal{R} \), v ∈ \( \mathbb{N} \), o ∈ \( \mathbb{N} \), rq ∈ \( \text{Message} \)) do
  return l = leader(v) ∧ pr = \( \langle \text{Prepare}, l, v, o, rq \rangle |\subseteq \langle (l, O, v | o, pr) \rangle \) ∧ hasValidCertificate(pr) ∧ isCorrectRequest(rq, . . .)

upon input call receive(pr = \( \langle \text{Prepare}, l, v, o, rq \rangle \)...)
if l ≠ r ∧ v = ostatus_r.nextStableView() ∧ inOrderWindow(o) ∧ isCorrectPrepare(pr, . . .) then
  omgs_r := omgs_r ∪ \{pr\}

upon internal task nextPrepared(pr = \( \langle \text{Prepare}, l, v, o′, rq \rangle \)...)
with
  ostatus_r.isStableFollower()
```

http://publikationsserver.tu-braunschweig.de/get/64440
\[ \text{upon internal call} \]
\[
\text{sendCommit}(v, o', \text{digest}(rq))
\]
\[
o\_prep_r := o'
\]
\[
\text{o\_act_r := o'}
\]

\[ \text{upon internal call} \]
\[
\text{sendCommit}(v \in \mathbb{N}, o \in \mathbb{N}, d \in \{\text{Digest} \ldots\})
\]
\[
\text{co} := (\text{COMMIT}, r, v, o, d)_{\forall (r, \omega, v) \in -}
\]
\[
\text{invoke tss, createIndependentCertificate}(\Omega, v | o, co)
\]
\[
\text{omsgs_r := omsgsr} \cup \{co\}
\]
\[
\text{output repgrpconn, send(co)}
\]

\[ \text{upon global call} \]
\[
\text{isCorrectCommit}(co, r' \in \mathbb{R}, v \in \mathbb{N}, o \in \mathbb{N}, d \in \{\text{Digest} \ldots\})
\]
\[
\text{return } r' \neq \text{leader}(v) \land co = (\text{COMMIT}, r', v, o, d)_{\forall (r', \omega, v) \in -} \land \text{hasValidCertificate(co)}
\]

\[ \text{upon input call} \]
\[
\text{receive(co = (COMMIT, r', v, o, d) \ldots)}
\]
\[
\text{if } r' \neq r \land v = \text{ostatus, stableView()} \land \neg \text{isCommitted(o)} \land \text{inOrderWindow(o)}
\]
\[
\text{\land isCorrectCommit(co, \ldots) then}
\]
\[
\text{omsgs_r := omsgsr} \cup \{co\}
\]

\[ \text{upon global call} \]
\[
\text{isCorrectCommittedCertificate}(O \subseteq \text{Message}, v \in \mathbb{N}, o \in \mathbb{N}, rq \in \text{Message})
\]
\[
\text{return } |O| \geq q \land (\exists pr \in O: \text{isCorrectPrepare(pr, \ldots, v, o, rq)}
\]
\[
\land (\forall m \in O: m = pr \lor \text{isCorrectCommit(m, \ldots, v, o, digest(rq))})
\]

\[ \text{upon internal task} \]
\[
\text{nextCommitted(rq = \langle \text{REQUEST}, c, t, \text{second} \ldots\rangle)}
\]
\[
\text{with}
\]
\[
o' = o\_comm_r + 1
\]
\[
\exists O \subseteq \text{omsgs_r: isCorrectCommittedCertificate}(O, \ldots, o', rq)
\]
\[
\text{do}
\]
\[
\text{markCommitted(o')}
\]
\[
\text{output em, executeRequest(o', rq)}
\]

\[ \text{upon input call} \]
\[
\text{forwardOrderWindow(o' \in \mathbb{N})}
\]
\[
\text{asserts}
\]
\[
o' > o\_base_r
\]
\[
\text{do}
\]
\[
o\_base_r := o'
\]
\[
\text{discardOrderMessages()}
\]
\[
\text{if } \neg \text{isCommitted(o')} \text{ then}
\]
\[
\text{markCommitted(o')}
\]

\[ \text{upon internal call} \]
\[
\text{discardOrderMessages()}
\]
\[
\text{omsgs_r := \{m \in omsgsr | \text{inOrderWindow(orderNumber(m))}\}}
\]
\[
\text{output repgrpconn, remove(\{m | (m = \langle \text{PREPARE} \ldots\rangle \lor m = \langle \text{COMMIT} \ldots\rangle) \land m \notin omsgsr\})}
\]

\[ \text{upon internal call} \]
\[
\text{markCommitted(o \in \mathbb{N})}
\]
\[
\text{asserts}
\]
\[
o > o\_comm_r
\]
\[
\text{do}
\]
\[
\text{if } o > o\_prep_r \text{ then}
\]
\[
o\_prep_r := o
\]
\[
o\_comm_r := o
\]

\[ \text{upon output call} \]
\[
\text{isCommitted(o \in \mathbb{N})}
\]
\[
\text{return } o \leq o\_comm_r
\]

\[ \text{upon output call} \]
\[
\text{lastCommitted()}
\]
\begin{verbatim}
107 return o_comm,
108
109 upon output call inOrderWindow(o ∈ °N) do
110 return o_base, < o ≤ o_max
111
112 upon output call orderWindowBase() do
113 return o_base,
114
115 upon output call obtainOrderingState(o_base ∈ °N) do
116 return \{m ∈ omsgs | orderNumber(m) > o_base\}
117
118 upon global call isCorrectOrderingState(I ⊆ Message, v ∈ °N, o_base ∈ °N) do
119 return ∀m ∈ I ∃o: (isCorrectPrepare(m, v, o, ·) ∨ isCorrectCommit(m, v, o, ·))
∧ o_base < o ≤ o_base + order wnd
120
121 upon input call installOrderingState(I ⊆ Message) do
122 asserts
isCorrectOrderingState(I, ostatus, stable View()) do
123 omsgs, := omsgs, ∪ \{m ∈ I | inOrderWindow(orderNumber(m))\}
124
125 upon output call createAbortState() → AbortState do
126 P := \{(\text{Prepare}, o, ·, o, ·) ∈ omsgs, | o - 1 = o_base, \vee (\text{Prepare}, o, ·, o - 1, ·) ∈ omsgs, \}
∧ (P, ostatus, current View()) \rightarrow o_act,
127
128 upon input call abort View(v' ∈ °N, m ∈ \{\ldots \}_r(\ldots)) do
129 invoke tss, createConsecutiveCertificate(\mathfrak{O}, v'[0, m)
130 o_act, := 0
131 ostatus, leave ViewFor(v')
132
133 upon global call isCorrectAbortState(\mathcal{B} = (P, tv_last) ∈ AbortState, v ∈ °N, v' ∈ °N, o_base ∈ °N, m ∈ \{\ldots \}_τ(\mathcal{O}, v' | 0, tv_last) do
134 return hasValidCertificate(m) \land isCorrectAbortState(b, v, v', o_base)
135
136 upon global call isCorrectAbortState(\mathcal{B} = (P, tv_last) ∈ AbortState, v ∈ °N, v' ∈ °N, o_base ∈ °N) do
137 return tv_last = (v, o_ctr)
∧ (∃o_max: isCorrectPrepareSet(P, v, o_base, o_max) \land o_max ≥ o_ctr)
138
139 upon global call isCorrectPrepareSet(P, v ∈ °N, o_base ∈ °N, o_max ∈ °N) do
140 return |P| = 0 \land o_max = o_base \lor 0 < |P| ≤ order wnd
∧ (\forall pr ∈ P \exists o: isCorrectPrepare(pr, v, o, ·))
∧ (∃o' ∈ °N: o_base < o' ≤ o_max \rightarrow \{\text{Prepare}, o, ·, o', ·\} ∈ P)
141
142 upon input call abort View(v' ∈ °N) do
143 invoke tss, forwardCounter(\mathfrak{O}, v'[0)
144 o_act, := 0
145 ostatus, leave ViewFor(v')
146
147 upon input call installAbortState(b = (P, tv_last) ∈ AbortState) do
148 asserts
isCorrectAbortState(b, ostatus, stable View(), \ldots) do
149 installOrderingState(P)
150
151 upon output call createViewTransition(v' ∈ °N, B ⊆ AbortState) do
152 P* := \{pr | \{P, ·\} ∈ B \land pr ∈ P\}
\end{verbatim}
if $|P'| = 0$ then
  return $\emptyset$
else
  $o_{\max} := \maxOrderNumber(P')$
  $P' := \{ \langle \text{Prepare}, r, v', o, m \rangle \mid o > o_{\base}, \langle \text{Prepare}, \cdot, o, m \rangle \in P' \}$
  for $o : o_{\base} < o \leq o_{\max}$ do
    $pr' := \{ \langle \text{Prepare}, r, v', o, m \rangle \mid pr' \in P' \}$
    invoke $\text{tss}_r.\text{createIndependentCertificate}(\emptyset, v'|o, pr')$
  return $P'$

upon global call $\text{isCorrectViewTransition}(P', v' \in \mathbb{N}, o_{\base} \in \mathbb{N}, B \subseteq \text{AbortState})$ do
  $P' := \{ pr \mid \langle P, \cdot \rangle \in B \land pr \in P \}$
  return $\text{isCorrectPrepareSet}(P', v', o_{\base}, \cdot)$

and $(\forall pr \in P' : pr = \langle \text{Prepare}, \cdot, o, m \rangle \land (o > o_{\base} \rightarrow \langle \text{Prepare}, \cdot, o, m \rangle \in P'))$

and $(\forall pr' \in P' : pr' = \langle \text{Prepare}, \cdot, o, m \rangle \land (\langle \text{Prepare}, \cdot, o, m \rangle \in P'))$

upon input call $\text{enterView}(v' \in \mathbb{N}, P' \subseteq \text{Message})$ do
  $ostatus_r.\text{enterView}(v')$
  if $|P'| = 0 \lor r \neq ostatus_r.\text{lastestLeader}()$ then
    $o_{\prep r} := o_{\base}$,
  else
    $o_{\prep r} := o_{\max}$
    $o_{\act r} := o_{\max}$
    $o_{\comm r} := o_{\base}$,
  output $\text{repgrpconn}_r.\text{remove}(\{ m | m = \langle \text{Prepare}, \cdot \rangle \lor m = \langle \text{Commit}, \cdot \rangle \})$
  installOrderingState($P'$)

upon internal call $\text{orderNumber}(m \in \text{Message})$
asserts
  $m = (\langle \text{Prepare}, \cdot, o, \cdot \rangle \lor \langle \text{Commit}, \cdot, o, \cdot \rangle)$
  do
    return $o$

upon internal call $\maxOrderNumber(I \subseteq \text{Message})$
return $\max(\{ o | m \in I \land o = \text{orderNumber}(m) \})$

upon internal call $\minOrderNumber(I \subseteq \text{Message})$
return $\min(\{ o | m \in I \land o = \text{orderNumber}(m) \})$
4.7 Execution

Listing 17: Specification of the service execution.

```plaintext
1 module Replica::Execution

3 uses cm_r ∈ Replica::Client
4 uses km_r ∈ Replica::Checkpointing

6 def global ServiceSnapshot = S × (C → ReturnEntry)

8 upon init() do
9   svc_r := Service()
10  o_exec_r ∈ oN := 0

12 upon input call executeRequest(o ∈ oN, rq = ⟨REQUEST, c, t, svccmd⟩) do
13   o = o_exec_r + 1
14   if ¬cm_r.isRequestExecuted(rq) then
15     svcret := invoke svc_r.invoke(c, svccmd)
16     output cm_r.requestExecuted(rq, svcret)
17     o_exec_r := o
18     output km_r.stateReached(o)

22 upon output call createSnapshot() do
23   s := input svc_r createStateSnapshot()
24   retvals := input cm_r.createReturnValueMapSnapshot()
25   return ⟨s, retvals⟩

27 upon input call installSnapshot(o_exec ∈ oN, ⟨s, retvals⟩ ∈ ServiceSnapshot) do
28   invoke svc_r.installStateSnapshot(s)
29   invoke cm_r.installReturnValueMap(retvals)
30  o_exec_r := o_exec
```

http://publikationsserver.tu-braunschweig.de/get/64440
4.8 Checkpointing Protocol

Listing 18: Specification of the checkpointing protocol.

module Replica_r::Checkpointing
uses om_r ∈ Replica_r::Ordering
uses em_r ∈ Replica_r::Execution

def global
CheckpointState def \("N \times \) ServiceSnapshot
CheckpointSet def \{ X|\exists o \forall ck \in X: ck = \langle CHECKPOINT, \cdot, o', \cdot \rangle \land o = o' \}
ProvenCheckpoint def CheckpointSet \times \) ServiceSnapshot

let
chkpt_int ∈ \("N\), chkpt_int ≤ orderWindowSize()
retentWnd ∈ \("N\), retentWnd < orderWindowSize()

upon global call checkpointBase(o)
return if o < retentWnd then 0 else o – retentWnd

upon global call checkpointNumber((o, s*) ∈ CheckpointState) do
return o

upon global call checkpointState((o, s*) ∈ CheckpointState) do
return s*

upon global call checkpointNumber(K ∈ CheckpointSet)
asserts
∃o \forall (CHECKPOINT, \cdot, o', \cdot) \in K: o = o'

upon init() do
kmsgs_r ⊆ Message := \{
chkptsr ⊆ CheckpointState := \{
a_stab_r ∈ \"N\) := 0

upon input call stateReached(o ∈ \"N\) do
if (o mod chkpt_int) = 0 then
createCheckpoint(o)

upon internal call createCheckpoint(o ∈ \"N\) do
ks := \{o, em_r.createSnapshot()\}
chkptsr := chkptsr ∪ {ks}
sendCheckpoint(ks)

upon internal call sendCheckpoint((o, s*) ∈ CheckpointState)
ck := \langle CHECKPOINT, r, a, digest(s*)\rangle_{r,\cdot,\cdot,0,0}
invoke tss_r.createConsecutiveCertificate(\(\mathbb{M}\), 0, ck)
kmsgs_r := kmsgs_r ∪ \{ck\}
output repgroupm_r.send(ck)

upon global call isCorrectCheckpoint(ck, r' ∈ \(\mathbb{R}\), o ∈ \"N\), d ∈ \{Digest\ldots\}) do
return ck = \langle CHECKPOINT, r', o, d, \rangle_{r',\cdot,\cdot,0,0} \land hasValidCertificate(ck)

upon input call receive(ck' = \langle CHECKPOINT, r', o, d\ldots\rangle) do
```plaintext
if isRelevantCheckpoint(ck') ∧ isCorrectCheckpoint(ck', ...) ∧ \[d': (\text{CHECKPOINT}, r', o, d') \in kmsgs_r ∧ d' \neq d\] then
  storeCheckpoint(ck')

upon internal call isRelevantCheckpoint((\text{CHECKPOINT}, r', o, \cdot)...) do
  return r' \neq r ∧ isRelevantCheckpoint(o)

upon internal call isRelevantCheckpoint(o ∈ \mathbb{N}) do
  o = \text{input} \_om_, orderWindowBase() ∨ \text{input} \_om_, inOrderWindow(o)

upon internal call storeCheckpoint(ck' = (\text{CHECKPOINT}, r', o, d)...) do
  kmsgs_r := kmsgs_r \setminus \{ck ∈ kmsgs_r | ck = (\text{CHECKPOINT}, r', o, \cdot)\}

kmsgs_r := kmsgs_r ∪ \{ck'\}

upon internal call discardCheckpoints() do
  o_base := \text{input} \_om_, orderWindowBase()
  kmsgs_r := kmsgs_r \setminus \{(\text{CHECKPOINT}, \cdot, o, \cdot) ∈ kmsgs_r | o < o_base\}
  \text{output} repgrpconn, remove(\{ck | ck = (\text{CHECKPOINT}, \cdot, o, \cdot) ∧ ck \notin kmsgs_r\})

upon global call isCorrectCheckpointCertificate(K \subseteq \text{Message}, o ∈ \mathbb{N}, d ∈ \{\text{Digest} \ldots\}) do
  return o = 0 ∧ |K| = 0 ∨ |K| > f ∧ (∀ck ∈ K: isCorrectCheckpoint(ck, o, d))

upon internal task checkpointStable(o' ∈ \mathbb{N}, d ∈ \{\text{Digest} \ldots\}) with
  \exists s' : digest(s') = d ∧ (o', s') ∈ chkpts_r
  \exists K' \subseteq kmsgs_r : isCorrectCheckpointCertificate(K', o', d)
  (∀K \subseteq kmsgs_r \exists o : isCorrectCheckpointCertificate(K, o, \cdot) → o ≤ o')
  do
    forwardWindow(o')

upon internal call forwardWindow(o' ∈ \mathbb{N}) do
  o_stab_r := o'
  chkpts_r := chkpts_r \setminus \{(o, \cdot) ∈ chkpts_r | o < o'\}
  if checkpointBase(o') > input \_om_, orderWindowBase() then
    \text{invoke} \_om_, forwardOrderWindow(checkpointBase(o'))
  discardCheckpoints()

upon global call isCorrectProvenCheckpoint(pk, o ∈ \mathbb{N}, s^* \in \text{ServiceSnapshot}) do
  return pk = (C, s^*) ∈ \text{ProvenCheckpoint} ∧ isCorrectCheckpointCertificate(C, o, digest(s^*))

upon output call stableCheckpointNumber() do
  return o_stab_r

upon internal call hasStableCheckpoint() do
  return o_stab_r > 0

upon internal call stableCheckpoint() → \text{ProvenCheckpoint}
asserts
  hasStableCheckpoint()
  do
    (o, s^*) ∈ chkpts_r | o = o_stab_r
    K := \{(\text{CHECKPOINT}, \cdot, o, digest(s^*))... ∈ kmsgs_r\}
  return (K, s^*)

upon output call stableCheckpointCertificate() → \text{CheckpointSet} do
  if ~hasStableCheckpoint() then
    return Ø
```

33
else
\langle K, s^* \rangle := \text{stableCheckpoint}()

\text{return } K

\text{upon output call obtainCheckpointingState}(o_{\text{stab}} \in \mathbb{N}, o_{\text{comm}} \in \mathbb{N}) \text{ do}
\text{if } o_{\text{comm}} < \text{input om}_{\text{r}.\text{orderWindowBase}}() \text{ then}
\text{return } \text{stableCheckpoint}()
\text{else if } o_{\text{stab}} < o_{\text{stab}_{r}} \text{ then}
\text{return } \text{stableCheckpointCertificate}()
\text{else}
\text{return } \emptyset

\text{upon global call isCorrectCheckpointingState}(ks, o_{\text{stab}} \in \mathbb{N}) \text{ do}
\text{return } ks = \emptyset
\text{\lor } ks \in \text{CheckpointSet} \land (|ks| = 0 \text{ \lor } \text{isCorrectCheckpointCertificate}(ks, o_{\text{stab}}, .))
\text{\lor } \text{isCorrectProvenCheckpoint}(ks, o_{\text{stab}}, .)

\text{upon input call installCheckpointingState}(ks) \text{ asserts}
\text{isCorrectCheckpointingState}(ks, . . . ) \text{ do}
\text{if } ks \in \text{CheckpointSet} \text{ then}
\text{installCheckpointCertificate}(ks)
\text{else if } ks = (K, s^*) \land \text{input om}_{\text{r}.\text{isCommitted}}(\text{checkpointNumber}(K)) \text{ then}
\text{a := checkpointNumber(K)}
\text{output om}_{\text{r}.\text{installSnapshot}}(a, s^*)
\text{forwardWindow}(o)
\text{createCheckpoint}(o)
\text{installCheckpointCertificate}(K)

\text{upon input call installCheckpointCertificate}(K \in \text{CheckpointSet}) \text{ asserts}
\exists o: \text{isCorrectCheckpointCertificate}(K, o, .) \text{ do}
\text{if isRelevantCheckpoint}(o) \text{ then}
\text{for all } ck \in K \text{ do}
\text{if isRelevantCheckpoint}(ck) \text{ then}
\text{storeCheckpoint}(ck)
\text{if } o > o_{\text{stab}_{r}} \text{ then}
\text{check checkpointStable}()}
4.9 View-Change Protocol


```haskell
module Replica_r::ViewChange

uses cm_r ∈ Replica_r::Client
uses om_r ∈ Replica_r::Ordering
uses km_r ∈ Replica_r::Checkpointing
uses sm_r ∈ Replica_r::StateTransfer

upon init() do
  vstatus_r := ViewStatus(r)
  v_stab_r def ≡ vstatus_r.stableView()
  v_cur_r def ≡ vstatus_r.currentView()
  vmsgs_r ⊆ Message := {}

upon internal call stableViewCertificate() do
  asserts
  v_stab_r > 0
  do
    return ⟨New-View, ·, v_stab_r, . . .⟩ ∈ vmsgs_r

upon internal call suspectsAcceptedLeader() do
  return cm_r.suspectsLeader()

upon internal call isLeaderSuspectedByQuorum() do
  return ∃ V ⊆ vmsgs_r : |V| > f ∧ (∀ vc ∈ V : v_to > v_cur_r ∧ isCorrectViewChange(vc, ·, v_to))

upon internal task acceptedLeaderSuspected() with
  estatus_r.isInStableView()
  ∧ (suspectsAcceptedLeader() ∨ isLeaderSuspectedByQuorum())
  do
    leaveViewFor(v_cur_r + 1)

upon internal call leaveViewFor(v' ∈ *N) do
  v := v_cur_r
  l := leader(v)
  K := input km_r.stableCheckpointCertificate()
  b := input om_r.createAbortState()
  vc := ⟨View-CHANGE, r, v_stab_r, v', K, b⟩...
  if nv_timeout_r.isRunning() then
    nv_timeout_r.stop()
  output om_r.abortView(v', vc)
  vstatus_r.leaveViewFor(v')
  storeViewChange(vc)
  output repgrpconn_r.send(vc)
  if v = v_stab_r then
    output sm_r.startStateRequestTask()

upon global call isCorrectViewChange(vc, r' ∈ R, v_from ∈ *N, v_to ∈ *N) do
  return ∃ K ⊆ Message ∃ b ∈ AbortState ∃ o_base, o_stab ∈ *N:
  vc = ⟨View-CHANGE, r', v_from, v_to, K, b⟩ . . . ∧ v_from < v_to
  ∧ isCorrectCheckpointCertificate(K, o_stab, -)```

35
\[ \land o\_base = checkpointBase(o\_stab) \]
\[ \land isCorrectAbortState(b, v\_from, v\_to, o\_base, vc) \]

upon input call receive(\(v' = \{\text{VIEW-CHANGE}, r', v\_from, v\_to, K, b\}\)) do
  if isRelevantViewChange(\(v'\)) \& isCorrectViewChange(\(v'\)) then
    installViewChange(\(v'\))

upon internal call isRelevantViewChange(\(v' = \{\text{VIEW-CHANGE}, r', v\_from, v\_to, \ldots\}\)) do
  return \(r' \neq r \land (v\_to > v\_cur_r \lor \neg uStatus, isInStableView \land v\_to = v\_cur_r)\)

upon internal call storeViewChange(\(v' = \{\text{VIEW-CHANGE}, \ldots, v\_to, \ldots\}\)) do
  \(vmsgs_r := vmsgs_r \cup \{v'\}\)
discardViewChanges()

upon internal call discardViewChanges() do
  if vstatus, isInStableView() then
    \(V^{\text{new}} := \emptyset\)
  else
    \(V^{\text{new}} := \{vc \in vmsgs_r \mid vc = \{\text{VIEW-CHANGE}, \ldots, v\_cur_r, \ldots\}\}\)
  \(V^{\text{max}} := \{vc\_\max \in vmsgs_r \mid vc\_\max = \{\text{VIEW-CHANGE}, r', \ldots, v\_\max, \ldots\}\}\)
  \(\land v\_\max > v\_cur_r\)
  \(\land (\exists vc \in vmsgs_r : vc = \{\text{VIEW-CHANGE}, r', \ldots, v, \ldots\} \rightarrow v \leq v\_\max)\)
  \(VS := \{\langle v, V' \rangle \mid V' \subseteq vmsgs_r \land v > v\_stab\}
  \(\land isCorrectViewChangeCertificate(V', v\_stab, v)\)
  \(\land (V \subseteq vmsgs_r)\)
  \(\land isCorrectViewChangeCertificate(V, v\_stab, v) \rightarrow |V| \leq |V'|)\)
  if \(V^{\text{cur}} \leq v\_stab + 1\) then
    \(V^{\text{cur}} := V'\)
  else if \(\langle v\_cur_r, V' \rangle \in VS\) then
    \(V^{\text{cur}} := V'\)
  else
    \(V^{\text{cur}} := V' \land \{v\_cur_r - 1, V'\} \in VS\)
  if \(|VS| = 0\) then
    \(V^{\text{max}} := \emptyset\)
  else
    \(V^{\text{max}} := V' \land \{v\_\max, V'\} \in VS \land (\forall (v, \ldots) \in VS : v \leq v\_\max)\)
  \(vmsgs_r := vmsgs_r \setminus \{vc \in vmsgs_r \mid vc = \{\text{VIEW-CHANGE}, \ldots\}\}\)
  \(\land vc \notin (V^{\text{new}} \cup V^{\text{max}} \cup V^{\text{cur}} \cup V^{\text{max}})\)
  output reprgrpconn.remove(\(\{vc \mid vc = \{\text{VIEW-CHANGE}, \ldots\}\}\)
  \(\land vc \notin (V^{\text{new}} \cup V^{\text{max}} \cup V^{\text{cur}} \cup V^{\text{max}})\))

upon input call installViewChange(\(v' = \{\text{VIEW-CHANGE}, v\_from, v\_to, K, b\}\)) do
  storeViewChange(\(v'\))
  abortStateReceived(v\_from, K, b)

upon internal call abortStateReceived(v \in \(\mathbb{N}\), K \in \text{CheckpointSet}, b \in \text{AbortState}) do
  output km, installCheckpointCertificate(K)
  if v = v\_stab, then
    output om, installAbortState(b)

upon global call isCorrectViewChangeCertificate(V \subseteq \text{Message}, v\_from \in \(\mathbb{N}\), v\_to \in \(\mathbb{N}\)) do
  return \(|V| \geq q \land (\forall v, v\_from \in \mathbb{N} : v\_from \subseteq v\_from \land isCorrectViewChange(\langle v, v\_from, v\_to \rangle)\)

upon internal call isInOrderWindow(K \subseteq \text{Message})
  return checkpointBase(checkpointNumber(K)) \leq \text{input om, orderWindowBase}()
upon internal call isInOrderWindow($V \subseteq Message$, $A \subseteq Message$)
return checkpointBase(maxCheckpointNumber($V, A$)) \leq \text{input om}_{v \text{orderWindowBase}}() 

upon internal call maxCheckpointNumber($V \subseteq Message$, $A \subseteq Message$) do
return $max\{\text{checkpointNumber}(K) \mid \langle \text{VIEW-CHANGE}, \cdot, \cdot, K, \ldots \rangle \in V \lor \langle \text{NEW-VIEW-ACK}, \cdot, \cdot, K, \cdot \rangle \}$ 

upon internal task newViewExpected()
with
\neg \text{vstatus}, \text{isInStableView}() \land l' = \text{leader}(v_{cur}) \land l' \neq r
\exists V \subseteq \text{vmsgs}, : \text{isCorrectViewChangeCertificate}(V, v_{stab}, v_{cur},)
\neg \text{nv_timeout}, \text{isRunning}()
do
\text{nv_timeout}, \text{start}(l')

upon internal call suspectsDesignatedLeader() do
return \text{nv_timeout}, \text{isExpired}()

upon internal task designatedLeaderSuspected()
with
\neg \text{vstatus}, \text{isInStableView}() \land (\text{suspectsDesignatedLeader()} \lor \text{isLeaderSuspectedByQuorum}())
\exists V \subseteq \text{vmsgs}, : \text{isCorrectViewChangeCertificate}(V, v_{stab}, v_{cur},)\land \text{isInOrderWindow}(V, \emptyset)
do
installAbortStates(V)
leaveViewFor(v_{cur}, + 1)

upon internal task viewMissed($v' \in \mathbb{V}$)
with
\exists V \subseteq \text{vmsgs}, \exists v > v_{cur}, \land \text{isCorrectViewChangeCertificate}(V, v_{stab}, v')
\land \text{isInOrderWindow}(V, \emptyset)
do
installAbortStates(V)
leaveViewFor(v')

upon internal call installAbortStates($V \subseteq Message$)
for all $vc \in V$ do
vc := $\langle \text{VIEW-CHANGE}, \cdot, v_{from}, \cdot \cdot \cdot, b \rangle$
if $v_{from} = v_{stab}$, then
output om_{v \text{installAbortState}}(b)

upon global call isCorrectNewViewCertificate($V \subseteq Message$, $A \subseteq Message$, $v_{from} \in \mathbb{N}, v_{to} \in \mathbb{N}$)
do
return isCorrectViewChangeCertificate($V, v_{from}, v_{to}$)
\land (\forall a : \text{isCorrectNewViewAck}(na, \cdot, v_{from})
\land |\{r' | \langle \text{VIEW-CHANGE}, r', v_{from}, \ldots \rangle \in V \lor \langle \text{NEW-VIEW-ACK}, r', \ldots \rangle \in A\}| > f

upon internal task newLeaderViewReady()
with
\exists v' \geq v_{cur}, \exists V \subseteq \text{vmsgs}, : r = \text{leader}(v')
\land \text{isCorrectNewViewCertificate}(V, a, v')
\land \text{isInOrderWindow}(V, A)
do
establishView($v', V, A$)

upon internal call establishView($v' \in \mathbb{N}, V \subseteq Message$, $A \subseteq Message$) do
if $v' > v_{cur}$, then
output om_{v \text{establishView}}(v')
\( v_{\text{cur}} := v' \)
\( K := \text{input } km, \text{stableCheckpointCertificate}() \)
\( B := \text{allAbortStates}(V, A) \)
\( P' := \text{input } om, \text{createViewTransition}(v', B) \)
\( nv := (\text{NEW-View}, r, v', K, V, A, P') \)\( t^*(\text{\(v, v', nv\)}) \)
\( \text{invoke } \text{tss, createIndependentCertificate(}\mathfrak{I}, v', nv) \)
\( \text{storeNewView}(nv) \)
\( \text{enterView}(v', P') \)
\( \text{output } \text{repgrpconn}, \text{send}(nv) \)

\( \text{upon global call }\text{isCorrectNewView}(nv, v \in \mathbb{N}, K \subseteq \text{Message}) \) do
\( \text{return } \exists V, A, P', o_{\text{stab}}, o_{\text{base}}: \)
\( \quad \text{nv} = (\text{NEW-View, leader}(v), v, K, V, P') \)\( t^*(\text{\(v, v', nv\)}) \)
\( \quad \land \text{hasValidCertificate}(nv) \)
\( \quad \land \text{isCorrectNewViewCertificate}(V, A, v) \)
\( \quad \land \text{isCorrectCheckpointCertificate}(K, o_{\text{stab}}, -) \)
\( \quad \land o_{\text{stab}} \geq \max\text{CheckpointNumber}(V, A) \)
\( \quad \land o_{\text{base}} = \text{checkpointBase}(o_{\text{stab}}) \)
\( \quad \land \text{isCorrectViewTransition}(P', v, o_{\text{base}}, \text{allAbortStates}(V, A)) \)

\( \text{upon input call }\text{receive}(nv' = (\text{NEW-View, ...} . . .)) \) do
\( \text{if isRelevantNewView}(nv') \land \text{isCorrectNewView}(nv', ... \) then
\( \text{installNewView}(nv') \)

\( \text{upon internal call }\text{isRelevantNewView}(nv' = (\text{NEW-View, \text{\(v', \ldots\)}})) \) do
\( \text{return } v' > \text{latestKnownView()} \)

\( \text{upon internal call }\text{storeNewView}(nv' = (\text{NEW-View, \text{\(v', \ldots\)}})) \) do
\( \text{vmsgs} := \text{vmsgs} \cup \{nv'\} \)
\( \text{discardNewViews()} \)

\( \text{upon internal call }\text{discardNewViews()} \) do
\( \text{NV}_{\text{cur}} := \{nv_{\text{cur}} \in \text{vmsgs}, | \text{nv}_{\text{cur}} = (\text{NEW-View, \text{\(v, v_{\text{cur}}\)}, \ldots} . . . \}
\quad \land (v_{\text{cur}} = v_{\text{stab}}, \lor v_{\text{cur}} = \text{latestKnownView}()) \}
\( \text{vmsgs} := \text{vmsgs} \setminus \{nv \in \text{vmsgs}, | nv = (\text{NEW-View, ...})) \wedge nv \notin \text{NV}_{\text{cur}} \}
\( \text{output } \text{repgrpconn, remove}((nv | nv = (\text{NEW-View, ...}) \wedge nv \notin \text{NV}_{\text{cur}})) \)

\( \text{upon internal call }\text{installNewView}(nv' = (\text{NEW-View, l, v', K, V, P'})\ldots)) \) do
\( \text{storeNewView}(nv') \)
\( \text{for all } v \in V \) do
\( \text{if isRelevantViewChange}(vc) \text{ then} \)
\( \quad \text{installViewChange}(vc) \)
\( \text{output } km, \text{installCheckpointCertificate}(K) \)
\( \text{for all } (\text{VIEW-CHANGE, \ldots, K', \ldots}) \in V \)
\( \text{output } km, \text{installCheckpointCertificate}(K') \)

\( \text{upon internal call }\text{allAbortStates}(V \subseteq \text{Message}, A \subseteq \text{Message}) \) do
\( \text{return } \{b | (\text{\(V, V_{\text{\(A, A\)}}\)}} \in V \lor (\text{NEW-View-Ack, \ldots, b}) \in A \}

\( \text{upon internal call }\text{latestKnownView()} \) do
\( \text{return } \max\{v | (\text{\(V, V_{\text{\(A, A\)}}\)}} \in \text{vmsgs} \cup \{0\} \}

\( \text{upon internal task }\text{newFollowerViewReady}(nv = (\text{NEW-View, l, v', K', \ldots, P'})\ldots) \) with
\( \text{nv} \in \text{vmsgs}, \wedge v' = \text{latestKnownView()} \land v' > v_{\text{stab}}, \land \text{inOrderWindow}(K) \)
\( \text{do } \text{isbelated} := v' < v_{\text{cur}}, \)
if $v' > v_{\_cur}$, then
  output om$_n$.abortView($v'$)
  $v_{\_cur} := v'$
  enterView($v'$, $P'$)
if isbelated then
  sendNewViewAck()

upon internal call sendNewViewAck() do
  $K :=$ input $km_n$.stableCheckpointCertificate()
  $b :=$ input om$_n$.createAbortState($v_{\_cur}$)
  $na :=$ (New-View-Ack, $r, v_{\_cur}$, $K, b$)$_{(r, 0, 0, 0)}$
  invoke tss$_n$.createContinuingCertificate($0, 0, na$)
  storeNewViewAck($na$)
  output repgrpconn$_n$.send($na$)

upon global call isCorrectNewViewAck($na, r'$, $v, v_{\_cur}$, $K, b$) do
  $na :=$ (New-View-Ack, $r$, $v, K, b$)$_{(r, 0, 0, 0)}$
  $\exists K \subseteq$ Message $\exists b \in$ AbortState $\exists o_{\_stab} \in \mathbb{N}$:
  $\land$ hasValidCertificate($na$)
  $\land$ isCorrectCheckpointCertificate($K, o_{\_stab}$)
  $\land$ $o_{\_base} =$ checkpointBase($o_{\_stab}$)
  $\land$ isCorrectAbortState($b, v_{\_from}, v_{\_to}, o_{\_base}$)

upon input call receive($na' =$ (New-View-Ack, $r'$, $v$, $v_{\_cur}$)) do
  if isRelevantNewViewAck($na'$) $\land$ isCorrectNewViewAck($na'$) then
    installNewViewAck($na'$)

upon internal call isRelevantNewViewAck($na' =$ (New-View-Ack, $r'$, $v', v_{\_cur}$)) do
  return $\exists v' \geq v_{\_stab}$, $\land$ ($\forall v' \in$ vmsgs$_n' :$ $na =$ (New-View-Ack, $r'$, $v$, $v_{\_cur}$, $K, b$) $\rightarrow v' > v$)

upon internal call storeNewViewAck($na' =$ (New-View-Ack, $v'$, $v_{\_cur}$)) do
  vmsgs$_n :=$ vmsgs$_n \cup \{na\}$
  discardNewViewAcks()

upon internal call discardNewViewAcks() do
  $N_{\_max} := \{ na_{\_max} \in$ vmsgs$_n :$ $\land$ $v_{\_max} =$ (New-View-Ack, $r'$, $v_{\_max}$, $\ldots$)
  $\land v_{\_max} \geq v_{\_stab}$,
  $\land (\forall v' \in$ vmsgs$_n :$ $na =$ (New-View-Ack, $r'$, $v$, $v_{\_cur}$)) $\rightarrow v \leq v_{\_max}$
  $vmsgs :=$ vmsgs$_n \setminus \{ na \in$ vmsgs$_n :$ $\land na =$ (New-View-Ack, $v'$, $v_{\_cur}$) $\land na \notin N_{\_max}\}$
  output repgrpconn$_n$.remove($\{ na \mid$ $na =$ (New-View-Ack, $\ldots \land na \notin N_{\_max}\}$)

upon internal call installNewViewAck($na' =$ (New-View-Ack, $v'$, $K, b$)) do
  storeNewViewAck($na'$)
  abortStateReceived($v'$, $K, b$

upon internal call enterView($v' \in \mathbb{N}$, $P' \subseteq$ Message) do
  vstatus$_n$.enterView($v'$)
  discardViewChanges()
  discardNewView($v'$)
  discardNewViewAcks()
  if vstatus$_n$.isInStableView() then
    if ne_timeout$_n$.isRunning() then
      ne_timeout$_n$.stop()
  output sm$_n$.stopStateRequestTask()
  output om$_n$.enterView($v'$, $P'$)
  output cm$_n$.enterView($v'$)

39
276 upon output call createStateRequestContent() do
277 o_comm := input om,.lastCommitted()
278 o_stab := input km,.stableCheckpointNumber()
279 return (latestKnownView(), v_cur, o_stab, o_comm)
280
281 upon global call isCorrectStateRequestContent(sr) do
282 return sr = \{v_latest ∈ †N, v_cur ∈ †N, o_stab ∈ †N, o_comm ∈ †N\} ∧ o_stab ≤ o_comm
283
284 upon output call obtainState(\{v_latest, v_cur, o_stab, o_comm\}) do
285 if v_latest > v_stab, ∨ v_stab, ≠ latestKnownView()
286 return ∅
287 if v_latest < v_stab, then
288 nv := stableViewCertificate()
289 I := input om,.obtainOrderingState( input om,.orderWindowBase())
290 else
291 nv := ∅
292 I := input om,.obtainOrderingState(o_comm)
293 V := \\{vc ∈ vmsgs, | vc = \langle VIEW-CHANGE, ..., v_to, ...\rangle ... \∧ (v_to > v_latest ≥ v_cur ∨ v_to ≥ v_cur > v_latest)\}
294 A := \\{na ∈ vmsgs, | na = \langle NEW-VIEW-ACK, v, ..., v⟩ ... ∧ v ≥ v_latest\}
295 ks := input km,.obtainCheckpointingState(o_stab, o_comm),
296 return if nv ≠ ∅ \∨ |V| > 0 \∨ |A| > 0 \∨ ks ≠ ∅ \∨ |I| > 0
297 then \{v_stab, nv, V, A, ks, I\} else ∅
298
299 upon global call isCorrectState(state)
300 return state = (v, nv, V, A, ks, I) \∧ (∃K', o_base, o_stab:
301 \∧ (nv = ∅ \∨ isCorrectNewView(nv, v, K', ...) \∧ o_stab ≥ checkpointNumber(K'))
302 \∧ (∀vc ∈ V \exists v_to: isCorrectViewChange(vc, v, ... v_to) \∧ v_to ≥ v)
303 \∧ (∀na ∈ A \exists v': isCorrectNewViewAck(na, v, v') \∧ v' ≥ v)
304 \∧ o_base = checkpointBase(o_stab)
305 \∧ isCorrectCheckpointingState(ks, o_stab)
306 \∧ isCorrectOrderingState(I, v, o_base))
307
308 upon input call installState(\langle v, nv, V, A, ks, I\rangle)
309 output km,.installCheckpointingState(ks)
310 if isRelevantNewView(nv) then
311 installNewView(nv)
312 check newFollowerViewReady()
313 if v = v_stab, then
314 output om,.installOrderingState(I)
315 for all vc ∈ V do
316 if isRelevantViewChange(vc) then
317 installViewChange(vc)
318 for all na ∈ A do
319 if isRelevantNewViewAck(na) then
320 installNewViewAck(na)
4.10 State Transfer Protocol

Listing 20: Specification of the state-transfer protocol.

1 module Replica_r::StateTransfer

3 uses vm_r ∈ Replica_r::ViewChange

5 let state_period_r ∈ N

7 upon init() do
8  state_timer_r := Timer()

10 upon input call startStateRequestTask() do
11  output state_timer_r.schedule(state_period_r)

13 upon input call stopStateRequestTask() do
14  output state_timer_r.cancel()

16 upon internal task periodicallyRequestState() with
17  input state_timer_r.isExpired()
18  do
19    requestState()
20    state_timer_r.schedule(state_period_r)

23 upon input call requestState() do
24  src := input vm_r createStateRequestContent()
25  sr := ⟨State-Request, r, src⟩τ(r, M, 0, 0)
26  invoke tss_r.createConsecutiveCertificate(M, 0, sr)
27  output repgrpconn_r.remove({m | m = ⟨State-Request, . . .⟩})
28  output repgrpconn_r.send(sr)

30 upon global call isCorrectStateRequest(sr, r′ ∈ R, src) do
31  return sr = ⟨State-Request, r′, src⟩τ(r′, M, 0, 0)
32   ∧ hasValidCertificate(sr) ∧ isCorrectStateRequestContent(src)

33 upon input call receive(rs = ⟨State-Request, r′, src⟩) then
34  state := input vm_r.obtainState(src)
35  if state ≠ ∅ then
36    sendState(r′, state)

39 upon internal call sendState(r′ ∈ R, state) do
40  sm := ⟨STATE, r, state⟩τ(r, M, 0, 0)
41  invoke tss_r.createConsecutiveCertificate(M, 0, sm)
42  output repconns_r.remove({m | m = ⟨STATE, . . .⟩})
43  output repconns_r.send(sm)

45 upon global call isCorrectStateMessage(sm, r′ ∈ R, state) do
46  return sm = ⟨STATE, r′, state⟩τ(r′, M, 0, 0) ∧ hasValidCertificate(sm)
47   ∧ isCorrectState(state)

49 upon input call receive(sm = ⟨STATE, r′, state⟩) then
50  if isCorrectStateMessage(sm, r′, state) then
51    output vm_r.installState(state)
References


