

Formalizing chain replication

Mark Bickford and David Guaspari

September 27, 2006

The paper [1] defines a protocol for storage systems, called “chain replication,” that provides fault tolerance with high throughput and guarantees “strong consistency”: ([1],page 91)

- (i) operations to query and update individual objects are executed in some sequential order and
- (ii) the effects of update operations are necessarily reflected in results returned by subsequent query operations.

Fault tolerance is provided by replication, and individual servers are assumed to be “fail-stop”: ([1],page 93)

- each server halts in response to a failure rather than making erroneous state transitions, and
- a server’s halted state can be detected by the environment.

Schneider and van Renesse informally describe the chain replication protocol as follows: ([1],page 93)

In chain replication, the servers replicating a given object *objID* are linearly ordered to form a *chain*. The first server in the chain is called the *head*, the last server is called the *tail*, and request processing is implemented by the servers roughly as follows:

Reply Generation. The reply for every request is generated and sent by the tail.

Query Processing. Each query request is directed to the tail of the chain and processed there atomically using the replica of *objID* stored at the tail.

Update Processing. Each update request is directed to the head of the chain. The request is processed there atomically using replica of *objID* at the head, then state changes are forwarded along a reliable FIFO link to the next element of the chain (where it is handled and forwarded), and so on until the request is handled by the tail.

State is:

$Hist_{objID}$: **update request sequence**

$Pending_{objID}$: **request set**

Transitions are:

T1: Client request r arrives:

$Pending_{objID} := Pending_{objID} \cup \{r\}$

T2: Client request $r \in Pending_{objID}$ ignored:

$Pending_{objID} := Pending_{objID} - \{r\}$

T3: Client request $r \in Pending_{objID}$ processed:

$Pending_{objID} := Pending_{objID} - \{r\}$

if $r = \text{query}(objId, opts)$ **then**

reply according options $opts$ based on $Hist_{objID}$

else if $r = \text{update}(objId, newVal, opts)$ **then**

$Hist_{objID} := Hist_{objID} \cdot r$

reply according options $opts$ based on $Hist_{objID}$

Figure 1: State machine specification of protocol

T4: Client request $r \in Pending_{objID}$ processed no reply:

$Pending_{objID} := Pending_{objID} - \{r\}$

if $r = \text{update}(objId, newVal, opts)$ **then**

$Hist_{objID} := Hist_{objID} \cdot r$

Figure 2: Losing a response

Strong consistency thus follows because query requests and update requests are all processed serially at a single server (the tail).

In [1] the top-level specification of the protocol is expressed operationally, by saying that the replicated state machine should behave like the state machine described in figure 1. Figure 2 contains an additional transition, labeled T4, which is needed because failure of the tail can cause responses to be lost. The authors of [1] agree that omitting T4 from the original paper was an oversight.

We have formally proved the correctness of chain replication in NuPRL and, in fact, shown that it is more robust than claimed: it is correct under weaker failure assumptions. The correctness argument presented in the paper [1] covers a single failure (or, more generally, isolated failures—such that reconfiguration from a failure completes before the next one occurs). It also assumes a strict fail-stop model of failure, with perfect detection of failures. Our proof allows arbitrary failures, makes no assumptions about detection, and assumes only that nodes, while they live, obey certain safety properties.

Event logic We reason in the logic of events, a framework that can be straightforwardly expressed in any reasonable higher order logic, and can therefore be analyzed by many standard proof technologies (e.g., PVS, HOL, and Coq as well as NuPRL).

We write declarative specifications that state constraints relating sets of input and output events. To state our specifications we need only one concept from event logic, the notion of causal order: $<_c$ is a well-founded, transitive order on events.

If, intuitively, event e causes e' , then $e <_c e'$ (though the converse may not be true). Saying that $<_c$ is well-founded says that the search for the causes of an event cannot lead to an infinite regression. (Technically, this allows us to argue by induction on causal order.)

The logic of events naturally models distributed systems, so we do not assume that events have any global temporal order. However, if $e <_c e'$, then we can be assured that e did occur temporally before e' .

Formal parameters The specification defined by figures 1 and 2 contains some undefined terms that appear in the specification as formal parameters. The first three of these are:

- Types *Cmd* and *Resp*. When an event e issues a request, we associate with e a value in *Cmd* identifying what request was issued. Similarly, an event representing a response is associated with a value in *Resp*.

Cmd is partitioned into $\{Update, Query\}$.

- The map $isupdate : Cmd \rightarrow \mathbb{B}$, provides a convenient way to represent that partition.

The other two parameters formalize the two ways in which the phrase

reply according options *opts* based on

occurs in figure 1. One denotes the response to a query and the other denotes the response to an update. The response to an update is based on $Hist_{objID}$, the history of all update requests (including the current one), and the response to a query is based on the update history and the current query. Accordingly, we represent

- the response to an update by the function $\Delta : (Update)List \rightarrow Resp$
- the response to a query by the function $Q : (Update)List \times Query \rightarrow Resp$

Deriving a declarative specification The specification in terms of a global abstract state machine says that the input and output events must be related in a way that is consistent with the input/output behavior of the abstract machine. So that we can make refinement steps that are just logical implications between one set of constraints and another, we want a specification of the replicated

state machine that constrains the relation between input events E_{In} and output events E_{Out} directly.

Intuitively, the proximate cause of every output event is one input event: Each output event comes from a transition T3 in response to a request r in the set $Pending_{objID}$ and each $r \in Pending_{objID}$ was added to the pending set by transition T1, an input event. So we introduce a function

$$f : E_{\text{Out}} \rightarrow E_{\text{In}}$$

relating an output to its proximate cause and then, by examining figures 1 and 2, deduce constraints that it must satisfy.

Furthermore, each output event is a reply “based on $Hist_{objID}$ ”, and every update request in $Hist_{objID}$ was added by transition T3 or T4 and the update that was added came from the set $Pending_{objID}$. Thus, there is a function

$$hist : E_{\text{Out}} \rightarrow (E_{\text{Up}})\text{List}$$

that maps the output event to the list of input events that correspond to the sequence $Hist_{objID}$ in the state of the abstract machine at the transition T3 that generated the output event.

To state the constraints concisely, we establish some notation. For any $e \in E_{\text{In}}$, we let $\text{In}(e)$ be the request value that has arrived, and for $e' \in E_{\text{Out}}$, we let $\text{Out}(e')$ be the response value that is sent. We let E_{Up} be those events $e \in E_{\text{In}}$ whose value $\text{In}(e)$ is an update, i.e. $\text{In}(e) \in \text{Update}$, and E_{Qu} will be the input events whose value is a query, i.e. $\text{In}(e) \in \text{Query}$. If es is a list of events in E_{In} , then $\vec{\text{In}}(es)$ will be the list of requests from the events in es , i.e. $\text{map}(\text{In}, es)$.

We therefore deduce the following constraints on f and $hist$. For any output event $e \in E_{\text{Out}}$

$$f(e) \in E_{\text{Qu}} \Rightarrow \text{Out}(e) = Q(\vec{\text{In}}(hist(e)), \text{In}(f(e))) \quad (1)$$

$$f(e) \in E_{\text{Up}} \Rightarrow \text{Out}(e) = \Delta(\vec{\text{In}}(hist(e))) \quad (2)$$

$$f(e) \in E_{\text{Up}} \Rightarrow f(e) = \text{last}(hist(e)) \quad (3)$$

$$\forall e' \in hist(e). e' <_c e \quad (4)$$

Consistency constraint The crucial consistency property that the abstract state machine specification expresses is that the state variable $Hist_{objID}$ is monotonically increasing so that the $Hist_{objID}$ in an earlier state is a prefix of the $Hist_{objID}$ in a later state. Thus, we might want to specify that for $e_1, e_2 \in E_{\text{Out}}$, if $e_1 <_c e_2$ then $hist(e_1)$ is a prefix of $hist(e_2)$. This strong ordered consistency property could be true if the output events are the events at the replicated state machine that sent the responses, but if the output events are the arrivals of the responses at the clients, then unless we can guarantee that the responses arrive in the same order that they were sent we will not be able to ensure this strong ordered consistency. We will argue that it is sufficient to guarantee a formally

weaker consistency property: for any two output events, the history of one will be a prefix of the history of the other.

Notice that the specification in figure 1 does not preserve the ordering of input events. The requests from input events are added to the unordered set $Pending_{objID}$ and the other transitions choose requests from this set non-deterministically. This behavior of the abstract machine reflects the fact that inputs arrive at the replicated state machine from various clients and even requests from the same client may travel over different channels (this must happen when the head of the chain has changed between requests) so there is no ordering on input requests that can always be preserved.

Similarly, the output responses may go to various clients (a response to a single request might be broadcast to several receivers) and responses to a single client may travel over different channels (and this must happen when the tail of the chain changes between requests). Thus, it would be difficult for the replication protocol to guarantee the strong ordered consistency property even if all responses from a given node to an individual clients were sent on a single FIFO channel, and it would be impossible to guarantee ordered consistency without such an assumption. The end-to-end argument says that it will not pay to enforce a property that general network behavior may later invalidate, since in such a case the end users will be forced to check the property themselves anyway.

The formally weaker consistency property (using \sqsubseteq for prefix) is:

$$\forall e_1, e_2 : E_{\text{Out}}. \text{hist}(e_1) \sqsubseteq \text{hist}(e_2) \vee \text{hist}(e_2) \sqsubseteq \text{hist}(e_1) \quad (5)$$

This property allows the end clients to easily recover the strong ordered consistency property if the response functions Q and Δ simply include the length of the history (i.e. a sequence number) as part of the response. Clients may then reorder the responses they receive or ignore out of order responses and thus enforce the ordered consistency property if it is required.

Client view specification Gathering all these constraints together we can directly state the clients' view of the replicated state machine as a relation between input events and output events:

$$\begin{aligned} & \exists f : E_{\text{Out}} \rightarrow E_{\text{In}}, \text{hist} : E_{\text{Out}} \rightarrow (E_{\text{Up}})\text{List}. \forall e : E_{\text{Out}}. \\ & f(e) \in E_{\text{Qu}} \Rightarrow \text{Out}(e) = Q(\vec{In}(\text{hist}(e)), \text{In}(f(e))) \\ & \wedge f(e) \in E_{\text{Up}} \Rightarrow \text{Out}(e) = \Delta(\vec{In}(\text{hist}(e))) \\ & \wedge f(e) \in E_{\text{Up}} \Rightarrow f(e) = \text{last}(\text{hist}(e)) \\ & \wedge \forall e' \in \text{hist}(e). e' \leq_c e \\ & \wedge \forall e' : E_{\text{Out}}. \text{hist}(e') \sqsubseteq \text{hist}(e) \vee \text{hist}(e) \sqsubseteq \text{hist}(e') \end{aligned}$$

Protocol interface specification Typically, we will implement the protocol as a module, with an API through which it communicates to clients.

We specify this module as an instance of the client specification in which inputs and outputs are identified with events at the API. It is easy to show that if we connect E_{In} and E_{Out} to this API by any (possibly unreliable) network, the client view specification will be satisfied.

In the API we define the input events (which we continue to call E_{In}) to be the arrival of an update at the head of the chain or the arrival of a query at the tail of the chain and the output events (which we continue to call E_{Out}) to be the arrival of a request (update or query) at the tail of the chain. These decisions determine what f must be.

An event e that is the arrival of a query at the tail of the chain is both an input event and an output event. Thus, if e is a query event, $f(e) = e$. The constraint $f(e) = \text{last}(\text{hist}(e))$ defines $f(e)$ for non-query output events. The specification of the API becomes:

$$\begin{aligned}
& \exists \text{hist} : E_{\text{Out}} \rightarrow (E_{\text{Up}})\text{List}. \forall e : E_{\text{Out}}. \\
& \quad e \in E_{\text{Qu}} \Rightarrow \text{Out}(e) = Q(\vec{In}(\text{hist}(e)), \text{In}(e)) \\
& \quad \wedge e \notin E_{\text{Qu}} \Rightarrow \text{Out}(e) = \Delta(\vec{In}(\text{hist}(e))) \\
& \quad \wedge e \notin E_{\text{Qu}} \Rightarrow \text{hist}(e) \text{ is non-empty} \\
& \quad \wedge \forall e' \in \text{hist}(e). e' \leq_c e \\
& \quad \wedge \forall e' : E_{\text{Out}}. \text{hist}(e') \sqsubseteq \text{hist}(e) \vee \text{hist}(e) \sqsubseteq \text{hist}(e')
\end{aligned}$$

Transition T2 allows an input to be ignored. This is expressed in our specification because we do *not* require that all events in E_{In} be found in the range of f or in a list in the range of hist .

Transition T4 allows responses to be lost (even for updates that are added to hist). This is expressed in our model because we do *not* require that every prefix of $\text{hist}(e)$, for any output e , be $\text{hist}(e')$ for some other output e' .

Formal proof Our formal proof of the chain-replication protocol is accomplished by refining this specification to more detailed implementation constraints that logically imply it. A detailed journal-style description of our proof contains links to its formalization in NuPRL.

Our weaker failure assumptions can be informally expressed as follows:

- Failure detection is not assumed to be perfect. Nodes can be configured out of the chain for any reason.

The proof in [1] assumes that a node will not be removed from the chain unless it has stopped.

- Failures are not assumed to be isolated.

The proof in [1] implicitly assumes that between failures the system is able to reconfigure. Our proof allow failures to occur at any time.

References

- [1] Robbert van Renesse and Fred B. Schneider. Chain replication for supporting high throughput and availability. In *OSDI*, pages 91–104, 2004.