Testing and Verifying Chain Repair Methods for CORFU Using Stateless Model Checking

Stavros Aronis[1(✉)], Scott Lystig Fritchie[2], and Konstantinos Sagonas[1]

[1] Department of Information Technology, Uppsala University, Uppsala, Sweden
{stavros.aronis,kostis}@it.uu.se
[2] VMware, Cambridge, MA, USA
sfritchie@vmware.com

Abstract. CORFU is a distributed shared log that is designed to be scalable and reliable in the presence of failures and asynchrony. Internally, CORFU is fully replicated for fault tolerance, without sharding data or sacrificing strong consistency. In this case study, we present the modeling approaches we followed to test and verify, using Concuerror, the correctness of repair methods for the Chain Replication protocol suitable for CORFU. In the first two methods we tried, Concuerror located bugs quite fast. In contrast, the tool did not manage to find bugs in the third method, but the time this took also motivated an improvement in the tool that reduces the number of traces explored. Besides more details about all the above, we present experiences and lessons learned from applying stateless model checking for verifying complex protocols suitable for distributed programming.

1 Introduction

This work began, as is often the case, around a whiteboard where a group of engineers were discussing distributed protocols used in cloud systems. Diagrams for two particular protocols were drawn, one for Chain Replication (Sect. 2.1) and one for CORFU (Sect. 2.3), a recently proposed variant of Chain Replication. Both protocols have been studied in research papers, but at the heart of the whiteboard discussion were protocol extensions to repair data after a replica crash; an area of less scientific scrutiny, but of obvious importance to implementors.

The discussion started with one particular replica repair method, known to work well when used in the original Chain Replication [16]. CORFU [13] is similar, but not identical to Chain Replication, therefore warranting an investigation about whether the differences are significant enough to cause that particular method to break in some cases. The verdict of the whiteboard discussion was that, indeed, there exists an execution scenario that violates safety in CORFU, and this same scenario could not manifest when repairing replicas in a system using the original Chain Replication algorithm. A different method was therefore proposed, which would not suffer from that particular weakness. Was this method correct however? No such verdict could be reached at the whiteboard discussion, as is again often the case.

© Springer International Publishing AG 2017
N. Polikarpova and S. Schneider (Eds.): IFM 2017, LNCS 10510, pp. 227–242, 2017.
DOI: 10.1007/978-3-319-66845-1_15

A stateless model checking tool like Concuerror (Sect. 3) should, at least in principle, also be able to find the bug that was discovered at the whiteboard discussion for the first method, and was therefore tried by one of the engineers. After creating executable models of CORFU and the replica repair extension (Sect. 4), the bug was indeed found by the tool. A model for the second repair method was therefore created and tested. After a few iterations, the tool managed to find a scenario that showed that this method was also erroneous. The engineer shared this (un)fortunate discovery on Twitter[1] catching the attention of Concuerror's developers, who were intrigued by the tweet and contacted him for more information about that particular use of their tool and for his experiences.

A fruitful collaboration began. At the engineer's end, several variations of repair techniques were devised and modeled, with new flaws in them found quickly by the tool. Eventually, a technique emerged that appeared to be safe. At the other end, the developers of Concuerror used this case study as inspiration to design and implement an improvement to the partial order reduction techniques that the tool employs (Sect. 5.1) and to also evaluate how effective a particular search space bounding technique was for finding bugs.

In this paper, we retell the story, starting with an overview of Chain Replication and CORFU (Sect. 2), including the ideas related to chain repair, followed by a brief overview of stateless model checking and Concuerror (Sect. 3). In the same section, we also briefly describe the main ideas behind the partial order reduction and bounding techniques that Concuerror uses to make testing and verification more effective. We then describe the initially used model, starting from the correctness properties that should hold and explaining in detail the various parts of the model that are related to them (Sect. 4). The chain repair methods are then described together with performance results that show the time and effort involved to find bugs in these methods or verify their correctness. The paper continues by describing and justifying refinements that were applied to the model, as well as an improvement that was implemented in Concuerror to increase its effectiveness (Sect. 5). All these enabled Concuerror to verify the correctness of the final repair method. The paper ends by reviewing related work (Sect. 6) and offering some final remarks (Sect. 7).

2 Chain Replication

Chain Replication [16] is a variation of leader/follower replication that supports linearizable single objects. In this section, we first describe the basic Chain Replication algorithm, including how repair of a failed server can be performed after the server restarts. Then, we describe a variation of the algorithm, which is used by the CORFU distributed log [13], and finally explain how porting the same repair technique to CORFU can lead to problems (e.g., linearizability violations).

[1] @slfritchie: "I was all ready to have a celebratory "New algorithm works!" tweet. Then the DPOR model execution w/Concuerror found an invalid case. Ouch." (https://twitter.com/slfritchie/status/745863131407220737).

2.1 Basic Algorithm

In Chain Replication's leader/follower protocol, all replica servers are arranged in an ordered list of *head*, *middle*, and *tail* servers. The head server is the leader; all other servers are followers. Clients send update operations to the head server.

If the head server rejects an update operation, it sends an error back to the client. If the operation is accepted, the head server does not reply, but sends state update requests down the chain. Each follower server (if any) records the update requests to their respective local data stores and then forwards the requests downstream, in the same order they were received. After an update has been stored by the last server in the chain, the tail server sends a successful acknowledgment (ack) to the client. Thus, for a single update to a chain of length three, four messages are required: client → head, head → middle, middle → tail, and tail → client.

Clients send read-only operations to the tail server, which is also the linearization point for all replicas. If the tail server stores a value, then all other servers upstream in the chain must already store that value or a newer one.

Note that a chain of length one is a single server that acts in both head and tail roles.

2.2 Chain Repair

The Chain Replication paper [16] is clear about what is required to shorten a chain when a server crashes or is otherwise stopped. It also discusses how to reintroduce a crashed server back into the chain, but omits details that an implementor must be aware of to maintain Chain Replication's linearizable consistency guarantee.

A naïve repair method might take the following steps:

1. stop all surviving servers in the chain, e.g., $[S^a_{head}, S^b_{tail}]$,
2. copy S^b_{tail}'s update history to the server under repair S^c_{repair}, then
3. restart all servers with a chain configuration of $[S^a_{head}, S^b_{middle}, S^c_{tail}]$.

This offline repair method is easy, but sacrifices cluster availability. Online repair is desirable, but we also wish to preserve Chain Replication's property of linearizable reads by sending only one query to a chain member.

The Chain Replication repair technique used by HibariDB [7] starts a repair with a transition from chain $[S^a_{head}, S^b_{tail}] \Rightarrow [S^a_{head}, S^b_{tail}, S^c_{repair}]$, where S^c is the crashed server. Read queries ignore the server under repair; they are sent to the tail server as usual. Updates are sent to the head server and propagate down the entire chain; replies are sent by S^c_{repair}. While this intermediate chain configuration is in place, a separate process aynchronously copies missing data from S^b_{tail} to S^c_{repair}. When all missing history items have been copied to the server under repair, all servers in the chain enter read-only mode. A flush command is sent by the head to force all pending writes down the chain to the tail. When the corresponding ack from the tail is received by the head, then we know that all update log histories must be equal: $S^a_{head} = S^b_{tail} = S^c_{repair}$. Finally, the chain transitions to $[S^a_{head}, S^b_{middle}, S^c_{tail}]$, and then read-only mode is canceled.

2.3 Chain Replication in CORFU

The design of the CORFU system [13] uses Chain Replication with three changes, related to what we described so far. First, the responsibility for implementing replication is moved to the client. CORFU servers do not communicate with each other, so it is impossible for them to implement the original Chain Replication protocol. Instead, the replication logic is embedded in the client. Thus, for a single update to a CORFU chain of length three, six messages are involved, in three pairs between each of client $\leftrightarrow$ head, client $\leftrightarrow$ middle, and client $\leftrightarrow$ tail.

The second change is that CORFU's servers implement write-once semantics. Clients may not replace or overwrite a previously written value.

Third, CORFU builds upon standard Chain Replication by identifying each chain configuration by an epoch number. All clients and servers are aware of the epoch number, and all client operations include the epoch number. If a client operation contains a different epoch number, the operation is rejected by the server. A server temporarily stops service if it receives a newer epoch number from a client. When any participant detects a change of epoch, it can retrieve the new configuration from a dedicated cluster layout configuration service.

2.4 Chain Repair Techniques for CORFU

Since CORFU's servers do not communicate directly with each other as HibariDB's servers do, the "read-only mode + sync flush down the chain" technique used by HibariDB cannot be directly applied to CORFU. Consider a scenario where a chain is undergoing repair during epoch #5 and there exist two clients, C_w and C_r. We are interested in the value of some piece of data, which starts with an *old* value (i.e., not_written, since each key can only be written once). Client C_w is writing a *new* value to the cluster. This scenario is illustrated in Fig. 1.

epoch #5	S^a_{head} value=*new*	S^b_{tail} value=*old* or value=*new*	S^c_{repair} value=*old*
epoch #6	S^a_{head} value=*new*	S^b_{middle} value=*new*	S^c_{tail} value=*old* or value=*new*

Fig. 1. An epoch & chain configuration change while a *new* value is written to the chain.

While epoch #5 is in effect, reads are sent to server S^b, which is in the tail role. All read operations during epoch #5 will return either the *old* or *new* value. If a client can read the *new* value, then all later reads will also read the *new* value. However, client C_w should also write to S^c, which is beyond the current tail. This operation can unfortunately be delayed by the network.

The repairer is not influenced by the new value, and can therefore change the cluster configuration to epoch #6. A race condition becomes possible. In epoch #6, S^c will receive read queries because it has the tail role. Now our writing and reading clients can race: if C_w is too slow to complete the write—disregarding even that it also has a wrong epoch number—then C_r can read the *old* value from the new tail cluster. Back during epoch #5, it was possible to read the *new* value. If we can now read the *old* value in epoch #6, then it looks like the value has gone "backwards in time". Such time travel violates the linearizability property. It is exactly this race condition that was discovered at the whiteboard discussion in the story of the introduction.

HibariDB's repair technique works because the head server knows about all pending writes: the head sends its flush message down the chain, and a final ack sent by the tail is eventually received by the head. HibariDB also stops new writes during the transition process. When the flush's ack is received by the head, all servers have the same update log history.

In contrast, CORFU has no central coordinator like HibariDB's head server. Can we use a variation of this HibariDB's repair technique without also introducing direct server $\leftrightarrow$ server communication? Does a variation exist that does not require tracking the state of all writing clients to orchestrate their behavior?

Let us briefly overview a particular testing and verification technique and tool that we can employ to answer these questions.

3 Stateless Model Checking, Erlang, Concuerror and Bounding

The problem of verification and testing of distributed systems and their algorithms is difficult, since one must consider all the different ways in which the involved entities can interact. *Model checking* techniques can explore the state space of a program that implements such an algorithm systematically, verifying that each reachable state satisfies some given properties. However, applying model checking to programs of realistic size is problematic, as it entails capturing, encoding and storing a large number of states.

Stateless model checking [10], also known as *systematic concurrency testing*, avoids this obstacle by exploring the state space of a program without explicitly storing intermediate global states. A special run-time scheduler drives program execution, recording operations that can be affected by the interaction between involved entities. State capturing is not needed, because if all such operations are executed in the same order from the initial state, then any previously encountered state can be reached again. Thus the effort of testing and verification can focus only on those operations. Stateless model checking has been successfully implemented in tools such as VeriSoft [11], CHESS [14] and Concuerror [12]. The last tool is specific to programs written in Erlang.

Erlang is an industrially relevant programming language based on the actor model of concurrency [2]. In Erlang, actors are realized by language-level processes implemented by the runtime system instead of being directly mapped to operating system threads. Each Erlang process has its own private memory area (stack, heap and mailbox) and communicates with other processes via asynchronous message passing with *copying semantics*. Processes then consume messages using *selective receive*, i.e., they can select which message to pick from their mailbox using pattern matching. The use of message passing for inter-process communication, rather than shared memory, makes distribution transparent. It also makes Erlang suitable for modeling distributed systems. Erlang has all the ingredients needed for concurrency via message passing and most of the ingredients (e.g., reads and writes to data stored in shared ETS tables, etc.) needed for concurrent programming using shared memory.

The tool we will employ, Concuerror [4], is a stateless model checking tool for finding errors in Erlang programs or verifying their absence[2]. Given a program and a test to run, Concuerror uses a dynamic exploration algorithm to systematically explore the execution of the test under conceptually all process interleaving. To achieve this, the tool performs a code rewrite that inserts instrumentation at code points where processes can yield control back to the scheduler during their execution. The instrumentation that Concuerror uses is selective (i.e., it takes place only at points that involve process actions that inspect or update some concurrency-related primitive that accesses VM-level data structures that are shared by processes) and allows Concuerror to control the scheduling when the program is run, without having to modify the Erlang VM in any way. Concuerror supports the complete Erlang language and can instrument and test programs of any size, automatically including any libraries they use.

Since the number of global states that can be reached due to different scheduling decisions in stateless model checking can be exponential in the number of execution steps, systematic concurrency testing algorithms use techniques such as *partial order reduction (POR)* and *bounding* to reduce the size of the search space.

Partial Order Reduction. POR techniques define equivalence classes among traces, based on the happens-before relation between the operations that occur in them [9]. POR algorithms aim to explore just one trace in each such equivalence class. Reversing the order of execution for a pair of racing operations that exists in an explored trace is a simple way to obtain a trace that belongs to a different equivalence class. Dynamic POR techniques start by executing an arbitrary scheduling and then explore additional traces, justified by the existence of races between actually executed operations. The exploration continues 'by need', trying to examine a minimal number of traces. Several DPOR algorithms have been proposed, including the *Optimal-DPOR* algorithm [1], a provably optimal DPOR algorithm that Concuerror is using.

Bounding. Even when using POR techniques, the exploration needs to examine a lot of complex interleaving of processes, as a direct result of reversing every possible pair of racing instructions. Bounding techniques try to limit the complexity of the explored traces in order to expose bugs that are "shallower". In order to do that, they impose constraints on how processes can be scheduled. Exploration begins with a budget which is expended whenever such a scheduling constraint is violated.

Preemption bounding [15] limits the number of times the scheduler can preempt (i.e., interrupt) a process in order to run other processes. The justification is that common patterns of concurrency bugs require few scheduling constraints and these in turn

[2] More information about Concuerror is at http://parapluu.github.io/Concuerror.

can be related to few preemptions [3, 18]. Delay bounding [6] is another bounding technique that forces the scheduler to always schedule the first non-blocked process out of a total order of all processes. The bound here is the number of times this order can be violated. Concuerror employs *exploration tree bounding*, a bounding technique that restricts the number of times a DPOR algorithm can consider schedulings different from the "first" one. In implementations of stateless model checking with DPOR, the first scheduling that is explored is usually the same as the one chosen under preemption bounding: a round-robin scheduling, in which processes execute without preemptions until they block. Exploration tree bounding limits the number of times exploration can 'diverge' from that first scheduling, and essentially combines the benefits of Optimal-DPOR (i.e., never even start to explore a trace if one that belongs to the same equivalence class has been already explored) with some of the benefits of delay bounding.

Having described our platform we now move on to the description of our models.

4 Modeling CORFU

In this section, we describe our modeling approach for verifying the correctness of methods for chain repair suitable for CORFU. We first list the correctness properties that we are interested in. We continue by describing how we model a number of servers and clients of CORFU using Erlang, followed by how we model each of the chain repair methods we want to test/verify. Finally, we give a short initial evaluation of the modeling. This section gives a faithful account of the engineer's initial effort, before the developers of Concuerror were involved.

4.1 Correctness Properties

We are interested to verify that CORFU servers and clients do not suffer from scenarios such as the one described earlier as "a value traveling backwards in time during a chain repair". More formally, we want the following correctness properties to hold.

Immutability: Once a value has been written in a key, no other value can be written to it.

Linearizability: If a read operation sees a written value for some particular key, subsequent read operations for that key must also see the same value.

4.2 Initial Model

A high-level view of the CORFU system that is modeled is the following: A number of stable servers (one or two suffice) will undergo a chain repair procedure to have a single additional server added to their chain. Concurrently, two other clients will try to write two different values to the same key, while a third client will try to read the key twice.

We make some assumptions about the state prior to running a repair simulation. At some earlier time, all servers were connected in the cluster's single chain. Then one server crashed, causing the chain to be shortened. The procedure to shorten the chain is well-understood and known to be safe, so it is excluded from the model. We also use only a single key/value pair in the store, corresponding to a single address in the CORFU log, as the aforementioned correctness properties impose constraints on just a single key (i.e., log address) in the CORFU system. We assume that none of the servers in the chain had a value for the key before the crash. After the crash, we assume that the crashed server restarts with an empty local data store. The repairing process, as well as writer and reader clients are all assumed to be concurrent and freely interleaved; strict ordering of operations exists only within a particular client, e.g., between the two read operations performed by the reader or between the steps of the repairing process.

This model, which in the rest of the paper we refer to as the *Initial Model*, is sufficient to reveal bugs in two of the chain repair methods we tested. Refinements of the initial model will be described later (Sect. 5.2), when we present the effort that went into the verification of the third repair method.

Servers and Clients in the Model. All servers and clients of the CORFU system are modeled as Erlang processes. These processes exchange messages corresponding to requests sent by clients to the servers and the respective server replies, as well as notifications to a central coordinator. All processes are running concurrently, allowing all possible interleaving between events to occur. As mentioned, Concuerror's scheduler can switch between processes at *every* point where instrumentation is added, and this ability can mimic the effect of network delays at any point in our model and the resulting message reordering. We are not interested in lost messages.

The types of processes used in the model are the following:

1. Central coordinator. This is the top process of the model and is responsible for spawning and setting up every other process (servers and clients), monitoring when all the clients are done and collecting their results, using assertions to check the correctness properties, and doing final cleanups (i.e., shutting down the servers). It is used as a modeling convenience; no such coordinator exists in a CORFU system.
2. CORFU log servers. These processes mimic the protocol and behavior specified by the servers in the CORFU system. There may be two or three of these processes: one for the server under repair, and the rest representing the healthy chain.
3. The layout server process. This process offers the cluster layout configuration service mentioned earlier. A "layout" data structure normally determines the chain order for each segment of the CORFU distributed log. In our model we assume that the layout contains only a single chain, and that reads and writes are to a single key; other aspects of the full CORFU system's layout structure are out of scope. Each layout change moves the system to a new epoch.
4. CORFU reading client. This is a process that attempts to read data twice. It must never experience "time travel" behavior by witnessing a written value followed by a not_written value (i.e., linearizability violation). Also, it should never witness two different written values (i.e., immutability violation).
5. CORFU writing clients. We have two writer client processes in the model, each attempting to write a value different from that of the other and report back to the coordinator. At most one such client is permitted to succeed.
6. The data repair process. This process executes all steps required for copying data to the server under repair and lengthening the chain afterwards. The steps required were described in general in Sect. 2.4, and are described in more detail below.

Coordinator's Details. The model includes an initialization and shutting down phase in which the coordinator sets up the servers and waits for all clients to complete their execution before shutting down the servers. Shutting down the servers is not strictly necessary, since Concuerror is always able to reset the state of the system before starting new schedulings, but we include it since a "cleanup phase" is common in testing.

When the clients are done, they send a message back to the coordinator, including information about the results of their operations. Specifically for the writers, the coordinator uses these results to determine whether more than one write was successful, violating the immutability property from the writers' point of view. The coordinator also inspects whether the log is left at a consistent state, with either no value written to the key, or a singular value being written consistently to a prefix of the chain.

CORFU *Log and Layout Servers' Details.* Servers never initiate any communication and only respond to requests by clients. As explained earlier, log servers know the current epoch and will notify clients that are trying to communicate using a wrong epoch number. Log servers support read and write operations for keys as well as epoch (and layout) update operations, while the layout server supports layout read and update operations.

CORFU *Clients' Details.* Clients communicate with log servers directly to read or write data. Write operations are sent to every server in the chain, while read requests are sent to the tail server only. We assume that clients begin with knowledge of the healthy chain of servers. If a client request is answered with the information that their epoch is wrong, they communicate with the layout server to get an update and use this information consistently to continue their operation.

Valid replies to a write request are ok, meaning that the write was successful, or written, which denotes that the key already had a value. Valid replies to a read request are not_written, which denotes that no value exists, or {ok,Val} where Val is the value read. A client request may also be left incomplete, signaled by a starved reply: too many concurrent layout changes have interrupted the request. In our model, the retry limit is higher than the number of layout changes performed by the repairing process.

Repair Process' Details. The data repair process executes the following steps: First, it changes the layout to include the crashed server in some place in the chain, depending on the repair method, without changing the head or tail servers. At that stage, read operations are still sent to the tail server, ignoring the server under repair, even in cases where it will eventually be in the tail position. On the other hand, write operations must succeed in all servers (including the server under repair) to be considered successful.

Second, the data repair process copies data from the tail server to the server under repair. In the model, the repair process needs to copy a single key's worth of data. We know the identity of server of the data source (tail), the destination (repair), and the one data key that we need to sync; all are hard-coded into the repair process. The outcome of any race between the repairer and the regular writer processes is checked for correctness at the end of model execution by the coordinator.

Third and last, after a successful second phase, the layout is once more changed to include the repaired server in its final place in the chain.

We will test three repair methods, differing in where the recovered server is placed in the chain: the head, the tail or an intermediate position. In the last case, we will test a configuration with two initially healthy servers, in which the position of the repaired server will be just between them, as well as a configuration with only one healthy server, which we will have to "logically split in two" to make space for the server under repair.

4.3 Method 1: Add Repaired Server at End of Chain

Repair using the "end of chain" method is starting from a $[S^a_{head}, S^b_{tail}]$ layout, transitioning to a $[S^a_{head}, S^b_{tail}, S^c_{repair}]$ layout, doing the value copying from S^b and then changing layout again to shift S^b to a middle role and S^c to the new tail: $[S^a_{head}, S^b_{middle}, S^c_{tail}]$.

This method is vulnerable to the race condition described in detail in Sect. 2.4. If we find the same bug in our model, we have some confidence that Concuerror is indeed a suitable tool to investigate correctness of methods for chain repair.

4.4 Method 2: Add Repaired Server at Start of Chain

This second repair technique is a variation of the first. Instead of putting the server under repair at the end of the chain, we put it at the beginning. The chain's configuration during the middle epoch looks like this: $[S^c_{repair}, S^a_{head}, S^b_{tail}]$. A write operation during repair in this chain configuration must be sent to S^c and then propagate down the chain to the other servers. Reads are always served by the tail S^b and the repair value is also copied from there. A writer trying to communicate with server S^a after repair has started will be notified that this is no longer the head and will have to ask for a layout update.

4.5 Method 3: Add Repaired Server in the Middle

In the final technique, the server under repair is placed in the middle of the chain. Our intuition suggests that this should be a safe thing to do. The original Chain Replication protocol has no direct contact between a client and a server in the middle of the chain. There should be no opportunity for a reader client to witness a consistency violation.

For CORFU's variant of Chain Replication, the client does interact with middle servers: the client cannot act upon the effect of a write unless the update is successful at all servers in the chain, applied serially in the chain's order.

This method uses three epochs of chain configuration: (i) epoch #1: $[S^a_{head}, S^b_{tail}]$, (ii) epoch #2: $[S^a_{head}, S^c_{repair}, S^b_{tail}]$, and (iii) epoch #3: $[S^a_{head}, S^c_{middle}, S^b_{tail}]$.

There is only one small problem with this method. What if the healthy chain is of length one? How can we insert the repaired server into the middle of a too-short chain? The proposed solution is to split the single server of the healthy chain into two logical servers: a logical head and a logical tail. The data stores of the two logical roles have different implementations.

For the logical tail role, the data store remains the same as CORFU's normal disk-based store. The differences are applicable only in the context of the head role's store.

Table 1. Runs of the methods using bounded and unbounded exploration.

Method	Bounded exploration			Unbounded exploration		
	Bug?	Traces	Time	Bug?	Traces	Time
1 (Tail)	**Yes**	638	57 s	**Yes**	3 542 431	144 h
2 (Head)	**Yes**	65	7 s	**Yes**	389	26 s
3 (Middle)	**No**	1257	68 s	**No**	>30 000 000	>750 h

The logical head role's store is split into a conceptual RAM-based and a disk-based store. If a key is unwritten, the value of an update operation is first written to the RAM-based half of the store. Later, if and when the update reaches the logical tail role, the value is written to the disk-based half of the store and the key is deleted from the RAM store. If the repair process is interrupted for any reason, the RAM store is discarded, and the next epoch change will fall back to a chain containing only the healthy server.

4.6 An Evaluation of the Repair Methods on the Initial Model

Let us see where we are so far. Table 1 shows the experimental results of running each of the three methods on the initial model using a standard desktop and the current version of Concuerror. We run Concuerror in two modes: (i) using exploration tree bounding (we used a bound of at most 4) in order to check for bugs, and (ii) without bounding the exploration, i.e., using the tool for verification. We explain our findings below.

Method 1: Add Repaired Server at End of Chain. When this model is executed, Concuerror finds the linearization violation described earlier. The reader process sees the value written by a writer in the tail of epoch #2, but after the repair process is completed, and moves the servers to epoch #3 (without copying that value) the reader's second read runs ahead of the writer and finds a non-written entry in the added server, since the writer has not yet also written there. The bug is found quite fast (in under a minute) when using bounded exploration. In contrast, without a bound, many more traces are explored before the bug is found and the hunt lasts for several days.

Method 2: Add Repaired Server at Start of Chain. Concuerror finds a case analogous to the problem of Method 1, where trouble happens immediately after an epoch change. Two different bugs are detected, depending on whether bounding is used or not.

In bounded exploration, the buggy trace, which is found very fast, involves a process scheduling that permits both writer processes to write different values to S^a and S^c: one during epoch #1 and the other immediately after the transition to epoch #2. Thus, Concuerror finds that the log history invariant outlined in Sect. 2.1 is violated. Recall that CORFU's server implements a write-once store. CORFU's write-once enforcement means that nobody can overwrite or replace the conflicting value that is now in the middle of the chain at S^a. Similarly, the bad value written at S^c cannot be altered.

In unbounded exploration the bug found is different. One of the writers starts a write, but is interrupted, so the write never reaches the tail server. Then the repair process starts, notifying the servers in the chain about the new head. The writer finds out about the new head and starts the write again from the top of the chain. The server under repair is not initialized and reports that the writer knows a newer layout, so it repeatedly denies its requests until the writer starves. The remaining clients finish with the new server unwritten at the head and a value committed at the second position (by the writer's first attempt). At this state, any subsequent writer can immediately move the system to a bad state (succeeding with a different value on the new head, and failing at the second server since a value is already there).

This scenario is arguably fixable if the layout server notifies the repaired server (that is to become the new head) before the other servers in the chain. However, this fix is still vulnerable to the repaired server accepting a value to its unwritten entry, which the repairer will not see at the tail, just as before.

Method 3: Add Repaired Server in the Middle. For the third method, we used the model of transition from 1-to-2 servers. Concuerror's bounded exploration did not find any trace that violates either Chain Replication's invariants or CORFU's invariants. This result is encouraging, but full verification was not achieved: unbounded exploration ran for many days without exhausting the search space.

Let us summarize the results of our evaluation so far: (1) Concuerror was able to detect problems in buggy methods fairly quickly. (2) In the first method, bounding was crucial for finding the bug in reasonable time. (3) The third method could not be verified.

5 Optimization and Refinements

Since full verification of the third method was not possible with the initial model, we describe the actions we took to increase the effectiveness of our approach: an optimization of the tool and two refinements of the model. Both were direct results of our investigation of the traces explored by Concuerror.

5.1 Optimization: Avoid Reordering the Delivery of Unrelated Messages

One of the design choices of Erlang's message passing mechanism, namely the fact that at the point when a process receives a message the contents of its mailbox are checked in the order of their arrival, can lead to a very simple race scenario: If multiple messages can match a `receive` statement, the message placed first in the mailbox will be the one selected. If the order of delivery is different, `receive` will pick a different message.

To be sound, Concuerror detects such races and explores all possible orders of delivery for such messages. However, Concuerror's original implementation had not been optimized to detect cases where a `receive` statement is written in a way such that only particular messages can be received, regardless of the delivery of other messages. Instead, the tool treated any two messages that were delivered to the same process as "possibly racing".

In our model, once a client process C has executed its code, it has to notify the coordinator process with a {done,C,...} message. Even though the coordinator is written in a way such that each such message can only be received by one particular receive statement (using a particular C value), Concuerror originally explored all possible orders of delivering such messages. This introduces a multiplicative factor in the number of traces that need to be explored, which is factorial in the number of clients.

To avoid this unnecessary exploration, we extended Concuerror with the ability to take into account the receive patterns used when a message is received when determining which other messages are racing with that message's delivery. As a trivial example of the usefulness of the extension, we note that the extended version of Concuerror will not try different delivery interleavings for messages that are never retrieved from a process' mailbox. The technical aspects of the implementation are beyond the scope of this paper, but its benefits will become evident in the final evaluation.

5.2 Two Refinements of the Model

Conditional Read. In the initial model, the reader issues two read requests, with the intent to detect values that change or disappear. Either bug observation is possible only if the first read operation sees a value written (from either writer). There exist, however, cases where the first read is either observing the location as not yet written, or is starved altogether. Issuing another read request in such cases cannot expose any bugs and only results in exploring unnecessary traces when interleaving this second read request.

In order to avoid such unnecessary exploration, we have refined the reader client so that it only attempts a second read operation if such an operation can actually reveal bugs: namely only if the first read operation sees some written value.

Convert Layout Server to an ETS Table. A second refinement of the model is to simplify the modeling of the communication with the layout server. The reader and writer clients communicate with the layout server just to read epoch and layout information, and in the initial model this communication is implemented with messages. Concuerror, even with the optimization described in Sect. 5.1, must explore both orderings in which requests from different clients arrive to the layout server; the server's receive patterns should be able to handle any client's request.

To avoid reorderings of requests that are layout read operations, and therefore *commutable*, we changed the modeling of the layout server to instead use a shared memory location in the Erlang Term Storage area for the layout information. Concuerror's knowledge of operations that conflict with each other is precise enough to not treat read operations to such a location as racing. Therefore it does not need to reverse them and explore "the other" trace. There will of course still be races involving the layout server: the repairing process has to write to the same shared memory location when changing epochs, introducing races with any read requests.

5.3 Evaluation of the Effect of the Optimization and Refinements

Recall that the only two cases where Concuerror did not complete in reasonable time was when bounding was not used. More specifically, Concuerror took a lot of time to find a bug in Method 1 and could not verify the correctness of Method 3.

For Method 3, applying the optimization and the two refinements above is sufficient. With these changes, Concuerror can verify that the new model has no bugs in 48 h, after exploring 3 931 413 traces. We did not evaluate the effect of each change on its own, since the required time is significantly larger (e.g., not using the optimization of Sect. 5.1 with four clients sending done messages back to the supervisor, conceptually leads to the exploration of $4! = 24$ times as many traces).

Table 2. Evaluation of improvements applied on Method 1, without bounding.

Optimization (Sect. 5.1)	Refinements (Sect. 5.2)		Traces	Time
	Cond. read	ETS layout		
✗	✗	✗	3 542 431	144 h
✓	✗	✗	151 923	5 h 30 m
✗	✓	✗	3 787	6 m 20 s
✓	✓	✗	212	19 s
✗	✗	✓	1 059 043	29 h 40 m
✓	✗	✓	47 148	1 h 05 m
✗	✓	✓	5 239	5 m 20 s
✓	✓	✓	289	18 s

For Method 1, we show more detailed results in Table 2. The message delivery order optimization reduces the time to the first bug to 5 h 30 m (151 923 traces) and the reader refinement even more so: a bug is found in 6 m 20 s (3 787 traces). When used together, these improvements can find a bug in the method in just 19 s (only 212 traces are explored). The layout server refinement is not so effective on its own and its application slightly increases the number of traces when combined with the reader refinement. With all three changes, the traces are shorter (no back and forth communication with an extra server) and thus the bug is found slightly faster (in 18 instead of 19 s) even though slightly more traces (289) are explored.

6 Related Work

An approach similar to ours has been described in the presentation of the P# language [5], which is suitable for designing asynchronous systems modeled as state machines. This modeling is very appealing for systems such as CORFU and indeed the chain replication algorithm has been included in the evaluation of the language. However, the bug-finding capabilities of the P# runtime are based on either depth-first systematic testing (without POR or our improvements for message passing), or random testing (which cannot be used for verification). Moreover the focus of the evaluation of chain replication is not on chain repair methods.

A different approach, used in the verification of distributed databases (including ones based on chain replication), has been to write a rigorous formal specification of the system and then use techniques such as temporal logic [8] or proof assistants [17] to complete the verification, possibly extracting an executable implementation from the specification. In contrast, our technique is using a simple, directly executable simulation of the system and all safety properties are described as plain, non-sophisticated assertions.

7 Concluding Remarks

We have described our experiences from using stateless model checking to test the correctness of three repair methods for the Chain Replication algorithm used in CORFU. Using a fairly straightforward model written in Erlang, we were able to find bugs in the first two repair methods using Concuerror, some more quickly detectable after applying a simple bounding technique. In an attempt to verify the correctness of the third repair method, we also designed and implemented an optimization for Concuerror, based on a particular pattern found in Erlang programs, and two techniques for refining the model. These changes allowed us to verify the correctness of the third chain repair method in reasonable time.

Acknowledgments. This work was supported in part by the Swedish Research Council and carried out within the Linnaeus centre of excellence UPMARC, Uppsala Programming for Multicore Architectures Research Center.

References

1. Abdulla, P., Aronis, S., Jonsson, B., Sagonas, K.: Optimal dynamic partial order reduction. In: Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2014, pp. 373–384. ACM, New York (2014). doi:10.1145/2535838.2535845
2. Armstrong, J.: Erlang. Commun. ACM **53**(9), 68–75 (2010)
3. Burckhardt, S., Kothari, P., Musuvathi, M., Nagarakatte, S.: A randomized scheduler with probabilistic guarantees of finding bugs. In: Proceedings of ASPLOS, ASPLOS XV, pp. 167–178. ACM, New York (2010). doi:10.1145/1736020.1736040
4. Christakis, M., Gotovos, A., Sagonas, K.: Systematic testing for detecting concurrency errors in Erlang programs. In: Sixth IEEE International Conference on Software Testing, Verification and Validation (ICST 2013), pp. 154–163. IEEE Computer Society (2013)
5. Deligiannis, P., Donaldson, A.F., Ketema, J., Lal, A., Thomson, P.: Asynchronous programming, analysis and testing with state machines. In: Proceedings of the 36th PLDI, PLDI 2015, pp. 154–164 (2015). doi:10.1145/2737924.2737996
6. Emmi, M., Qadeer, S., Rakamarić, Z.: Delay-bounded scheduling. In: Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, pp. 411–422. ACM, New York (2011)
7. Fritchie, S.L.: Chain replication in theory and in practice. In: Proceedings of the 9th ACM SIGPLAN Workshop on Erlang, Erlang 2010, pp. 33–44. ACM, New York (2010). doi:10.1145/1863509.1863515
8. Geambasu, R., Birrell, A., MacCormick, J.: Experiences with formal specification of fault-tolerant file systems. In: IEEE International Conference on Dependable Systems and Networks With FTCS and DCC, DSN 2008, pp. 96–101. IEEE (2008)
9. Godefroid, P.: Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem. Springer-Verlag New York Inc., Secaucus (1996)

10. Godefroid, P.: Model checking for programming languages using VeriSoft. In: Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1997, pp. 174–186. ACM, New York (1997). doi:10.1145/263699.263717
11. Godefroid, P.: Software model checking: the VeriSoft approach. Form. Methods Syst. Des. **26**(2), 77–101 (2005). doi:10.1007/s10703-005-1489-x
12. Gotovos, A., Christakis, M., Sagonas, K.: Test-driven development of concurrent programs using Concuerror. In: Proceedings of the 10th ACM SIGPLAN Workshop on Erlang, Erlang 2011, pp. 51–61. ACM, New York (2011). doi:10.1145/2034654.2034664
13. Malkhi, D., Balakrishnan, M., Davis, J.D., Prabhakaran, V., Wobber, T.: From Paxos to CORFU: a flash-speed shared log. SIGOPS Oper. Syst. Rev. **46**(1), 47–51 (2012). doi:10.1145/2146382.2146391
14. Musuvathi, M., Qadeer, S., Ball, T., Basler, G., Nainar, P.A., Neamtiu, I.: Finding and reproducing heisenbugs in concurrent programs. In: Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation, OSDI 2008, pp. 267–280. USENIX Association, Berkeley (2008)
15. Qadeer, S., Rehof, J.: Context-bounded model checking of concurrent software. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 93–107. Springer, Heidelberg (2005). doi:10.1007/978-3-540-31980-1_7
16. van Renesse, R., Schneider, F.B.: Chain replication for supporting high throughput and availability. In: Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation, OSDI 2004, pp. 91–104. USENIX, Berkeley (2004)
17. Schiper, N., Rahli, V., van Renesse, R., Bickford, M., Constable, R.L.: Developing correctly replicated databases using formal tools. In: 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pp. 395–406. IEEE (2014)
18. Thomson, P., Donaldson, A.F., Betts, A.: Concurrency testing using controlled schedulers: an empirical study. ACM Trans. Parallel Comput. **2**(4), 23:1–23:37 (2016). doi:10.1145/2858651