Increasing Performance in Byzantine Fault-Tolerant Systems with On-Demand Replica Consistency*

Tobias Distler Rüdiger Kapitza
Friedrich–Alexander University Erlangen–Nuremberg
{distler,rrkapitz}@cs.fau.de

Abstract

Traditional agreement-based Byzantine fault-tolerant (BFT) systems process all requests on all replicas to ensure consistency. In addition to the overhead for BFT protocol and state-machine replication, this practice degrades performance and prevents throughput scalability. In this paper, we propose an extension to existing BFT architectures that increases performance for the default number of replicas by optimizing the resource utilization of their execution stages.

Our approach executes a request on only a selected subset of replicas, using a *selector* component co-located with each replica. As this leads to divergent replica states, a selector on-demand updates outdated objects on the local replica prior to processing a request. Our evaluation shows that with each replica executing only a part of all requests, the overall performance of a Byzantine fault-tolerant NFS can be almost doubled; our prototype even outperforms unreplicated NFS.

Categories and Subject Descriptors D.4.7 [*Organization and Design*]: Distributed Systems; C.4 [*Performance of Systems*]: Fault Tolerance

General Terms Design, Performance, Reliability

Keywords Byzantine Failures; Performance

1. Introduction

Today's information society heavily depends on computer-provided services. Byzantine fault tolerance (BFT) based on replicated state machines is a general approach to make these services tolerate a wide spectrum of faults, including hardware failures, software crashes, and malicious attacks.

*This work was partially supported by the German Research Council (DFG) under grant no. KA 3171/1.

EuroSys'11, April 10–13, 2011, Salzburg, Austria.

In general, the architecture of an agreement-based BFT system can be divided into two stages [Yin 2003]: *agreement* and *execution*. The agreement stage is responsible for imposing a total order on client requests; the execution stage processes the requests on all replicas, preserving the order determined by the agreement stage to ensure consistency.

Up to now, a number of protocol optimizations and architecture variants [Castro 1999, Correia 2004, Hendricks 2007, Kotla 2007; 2004, Wester 2009] have been proposed to improve the performance of both stages. In all these cases, the performance increase achieved is based on minimizing certain parts of the overhead introduced by BFT replication (e. g., request ordering [Correia 2004, Kotla 2007] or state-machine replication [Kotla 2004]). However, as all these systems ensure replica consistency by executing all requests on all replicas, the maximum throughput achievable for the fault-tolerant service is bounded by the throughput of a single replica; that is, the corresponding non–fault-tolerant unreplicated service.

In this paper, we present an extension to existing BFT state-machine–replication architectures that increases the upper throughput bound of an agreement-based BFT service (in the absence of faults) beyond the throughput of the corresponding unreplicated service. Our evaluation shows that our extension also reduces the response time under load. The approach exploits the fact that for systems tolerating f faults, $f + 1$ *identical* replies provided by different replicas prove a reply correct. In accordance with other authors [Hendricks 2007, Kotla 2007, Wester 2009, Wood 2011], we assume faults to be rare. Therefore, we execute each request on only a subset of $f+1$ replicas during normal-case operation. In case of a fault, additional replicas process the request.

The subset of replicas to execute a request is selected for each request individually, based on the service-state variables accessed by the request. In particular, we divide the service state into *objects* and assign each object to $f + 1$ replicas. With different objects being assigned to different subsets of replicas, request execution is distributed across all replicas. Assuming uniform object distribution and uniform object access, each replica only executes $\frac{f+1}{n}$ of all requests, with n being the number of replicas. As a result, the

upper throughput bound T of the overall system increases to $T' = \frac{n}{f+1}T$. For a standard agreement-based BFT architecture with $3f + 1$ execution replicas and $f = 1$, this doubles the throughput bound. Our evaluation of a Byzantine fault-tolerant version of NFS shows that for medium and high workloads the load reduction at the execution stage outweighs the overhead introduced by request ordering and state-machine replication. As a result, the BFT service outperforms its non–fault-tolerant unreplicated equivalent.

Replica states in our system are not completely identical at all times but may differ across replicas, depending on the requests a replica has executed. However, instances of the same object are consistent across all $f + 1$ assigned replicas as they all process requests accessing the object in the same order. As clients may send requests accessing objects assigned to different replicas, a *selector* module co-located with every service replica provides *on-demand replica consistency (ODRC)*. The selector ensures that all objects possibly read or modified by a request are consistent with the current service state by the time the request is executed; objects outside the scope of a request are unaffected and may remain outdated.

Applying ODRC reduces the costs of keeping replicas consistent to the extent actually needed in order to provide consistent replies to client requests. In consequence, this approach improves resource utilization of replica execution stages, as replicas for the most part only execute requests that produce replies the client really needs to make progress. While other systems [Distler 2011, Wood 2011] have been proposed that minimize the number of replicas by utilizing the idea of providing only $f + 1$ replies in the absence of faults, the main goal of ODRC is to increase performance for the default number of replicas (i. e., $3f + 1$ in traditional BFT systems, such as PBFT [Castro 1999]).

We show that ODRC can be integrated into existing agreement-based BFT state-machine–replication architectures by introducing a single module between the agreement stage and the execution stage of a replica. Apart from that, our approach solely relies on existing mechanisms. In particular, this paper makes the following contributions:

- It presents an extension to existing agreement-based BFT state-machine–replication architectures that improves performance for the default number of replicas using ODRC (Sections 3 and 4).
- It outlines service integration for ODRC for a Byzantine fault-tolerant NFS service (Section 6) and a BFT variant of ZooKeeper (Section 7).
- It evaluates the impact of ODRC in the absence as well as the presence of faults (Sections 6 and 7).

In addition, Section 2 presents our system model. Section 5 describes important optimizations. Section 8 discusses the implications of applying on-demand replica consistency. Section 9 presents related work, and Section 10 concludes.

2. System Model and Assumptions

This section presents our system model and defines the BFT system properties required for ODRC.

2.1 General System Model

We assume the standard system model used for BFT state-machine replication [Castro 1999, Kotla 2007; 2004, Rodrigues 2001, Yin 2003] that comprises the possibility of replicas and clients behaving arbitrarily. Nodes may operate at different speeds. They are linked by a network that may fail to deliver messages, corrupt and/or delay them, or deliver them out of order. Our system is safe under this asynchronous model. Liveness is ensured when the *bounded fair links* [Yin 2003] assumption holds. Nodes authenticate the messages they send to other nodes; we assume that an adversary cannot break cryptographic techniques.

Replicas implement a state machine [Schneider 1990] to ensure consistency and deterministic replies to client requests; the state machine consists of a set of variables S encoding its state and a set of commands C operating on them. The execution of a command leads to an arbitrary number of state variables being read and/or modified and an output being provided to the environment. Our approach requires deterministic state machines: for the same sequence of inputs every non-faulty replica must produce the same sequence of outputs. Thereby, the replicas have to be in an identical state between processing the same two requests. The use of deterministic state machines guarantees that all non-faulty replicas executing a client request answer with identical replies.

We divide the state S into *objects*; an object O comprises a set of up to $|S|$ state variables ($0 < |O| \leq |S|$), sizes of different objects may vary. Altogether, objects cover the whole state ($\bigcup O_i = S$). For simplicity, we assume objects to be disjoint ($O_i \cap O_j = \emptyset$).

2.2 Architectural Properties

We assume that the BFT system architecture separates agreement from execution [Yin 2003] (SEP); our approach does not require agreement stage and execution stage to be located on different hosts. In particular, the following BFT systems[1] are suitable to integrate ODRC: PBFT [Castro 1999], BASE [Rodrigues 2001], CBASE [Kotla 2004], SEP [Yin 2003], TTCB [Correia 2004]), and VM-FIT [Reiser 2007]. Note that this list is not intended to be exhaustive.

To emphasize the generality of our approach, we use an abstract view of a BFT system as a composition of replicas (consisting of agreement stage and execution stage) and voters (responsible for identifying correct replies); voting is usually performed by the client. Replicas and voters may both be subject to Byzantine faults. The properties discussed in the following refer to non-faulty replicas and voters.

[1] Some of the systems listed do not explicitly separate agreement from execution. However, their basic concepts allow a separation of agreement stage and execution stage to the extent required for ODRC.

2.2.1 Replicas

A non-faulty replica must provide the following properties in order to be suitable for applying ODRC:

R1 **Total Request Ordering:** The agreement stage inputs an arbitrary sequence of client requests (that may differ between replicas) and outputs a stable totally-ordered sequence of requests (identical across all replicas).

R2 **Request Execution:** Each replica in the execution stage inputs the totally-ordered sequence of requests and outputs a set of replies that is identical to the output of all other non-faulty replicas.

R3 **Reply Cache:** Each replica caches replies in order to provide them to voters on demand.

$R1$ and $R2$ are standard techniques to ensure replica consistency. Note that we make no assumptions on how the total ordering of requests ($R1$) is achieved; that is, we treat the agreement stage as a black box. Therefore, ODRC can be implemented in systems using the standard BFT protocol [Castro 1999, Kotla 2004, Rodrigues 2001, Yin 2003], as well as in systems relying on trusted components [Chun 2007, Correia 2004, Reiser 2007].

In general, BFT systems provide the reply cache ($R3$) to be able to resend replies in case of network problems in order to guarantee liveness. Replicas usually store only the last reply sent to each client to limit cache size. In this paper, we assume that the reply cache is hosted by the agreement stage [Yin 2003].

2.2.2 Voters

A non-faulty voter must provide the following properties in order to be suitable for applying ODRC:

V1 **Reply Verification:** The result to a client request is accepted as soon as the voter receives $f + 1$ identical replies (or reply digests) from different replicas.

V2 **Incomplete-Voting Notification:** All non-faulty replicas eventually learn about an incomplete voting (i. e., the voter is not able to collect $f + 1$ identical replies within a certain period of time).

$V1$ is a standard property of BFT systems as $f + 1$ identical replies are the proof for correctness in the presence of at most f faulty replicas. $V2$ covers an essential mechanism to guarantee liveness. In systems using the standard BFT protocol [Castro 1999, Kotla 2007; 2004, Rodrigues 2001, Yin 2003], for example, it protects clients against a faulty primary not forwarding requests. In particular, on the expiration of a voting timeout, a client multicasts the affected request to all replicas, potentially triggering a view change.

2.3 Service-State Checkpointing

ODRC requires a BFT system to provide a mechanism to checkpoint the service state as well as a mechanism to restore the service state based on a checkpoint. In particular, we assume the execution stage to provide an *object checkpoint function* that creates a snapshot of a state object. We also demand the execution stage to provide an *object update function* that updates an object based on a given checkpoint.

As most BFT systems [Castro 1999, Kotla 2007; 2004, Reiser 2007, Rodrigues 2001, Yin 2003] rely on periodic checkpoints to correct faulty replicas as well as to bring slow replicas up to date, both functions may be implemented using existing mechanisms. For example, all BASE-related systems [Kotla 2004, Rodrigues 2001, Yin 2003] implement a copy-on-write approach that only snapshots state objects modified since the last checkpoint. Furthermore, these systems also provide a function to selectively update state objects. Therefore, object checkpoint and object update functions that satisfy our requirements can be implemented without major implementation changes.

2.4 Request Analysis Function

We require requests to carry information about which objects they might read or modify during execution. To extract this information, we assume the existence of an application-specific *request analysis function (RAF)* that inputs a request req and outputs a set of objects:

$$Set_{<Object>}\ RAF(Request\ req)$$

For each request req, this function determines the maximal set of objects that might be accessed during execution; we assume each request to access at least one object. Note that the function can be implemented conservatively (i. e., it may return more objects than actually accessed) for services where a detailed analysis is too expensive. For the remainder of this paper, we state that a request *accesses* an object when the object is included in the set of objects returned by the request analysis function, independent of whether the request actually reads or modifies the object during execution.

In general, it may not be possible to specify a request-analysis function for arbitrary services as requests do not necessarily contain enough information to determine the state objects they access. However, most replicated BFT systems use similar functions to efficiently implement state-machine replication. For example, systems derived from BASE [Kotla 2004, Rodrigues 2001, Yin 2003] utilize information about state access of requests to determine state changes; CBASE [Kotla 2004] executes requests in parallel based, among other criteria, on the state objects they access.

3. Selective Request Execution

In this section, we present an extension to the common BFT state-machine–replication architecture that implements *selective request execution*. Instead of processing all requests on all replicas, a request is executed on only a subset of replicas, selected based on the objects accessed by the request. As a result, selective request execution reduces the load on a replica's execution stage. In this section, we assume the absence of faults; we drop this assumption in Section 4.

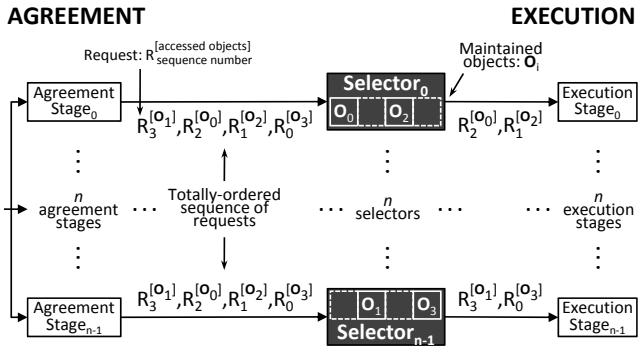


Figure 1. Based on the objects to be accessed, selectors selectively execute agreed requests on local replicas.

```
1 void insert(Request req) {
2   Set<Object> O_req = RAF(req);
3   ack_unmain(O_req \ O_main, req);
4   if (O_req ∩ O_main == ∅) {
5     R_store.enqueue(req);
6   } else {
7     update_objects(O_req \ O_main, req);
8     R_exec.enqueue(req);
9 } }
10
11 Request next_request() {
12   return R_exec.dequeue();
13 }
```

Figure 2. Selector algorithm

3.1 State Distribution

Selective request execution requires all replicas to host all state objects. However, each replica is only responsible for keeping an assigned subset of objects up to date. In particular, objects are distributed across the system so that each object is assigned to $f + 1$ replicas; for the remainder of this paper, we refer to an object assigned to a replica as a *maintained object*. As a result, each object is maintained on $f + 1$ replicas and *unmaintained* on all others. We assume that a replica is able to determine its set of maintained objects based on a global assignment relation (e. g., for $f = 1$ and $n = 4$, a replica with ID $r \in [0, n - 1]$ is assigned all objects with IDs o being $o \; mod \; 2 = r \; mod \; 2$; see Figure 1).

3.2 Selector

To apply selective request execution, we introduce a *selector* module between agreement stage and execution stage (see Figure 1). Each replica comprises its own selector that manages local request execution. Selectors of different replicas do not interact with each other, but rely on the same deterministic state machine (see Section 3.3) and operate on the same input; that is, the totally-ordered sequence of requests provided by the agreement stage ($R1$). As a result, all non-faulty selectors behave in a consistent manner. The selector provides the following lightweight interface:

```
void insert(Request req);
Request next_request();
void force_request(Request req);
```

Relying on the two functions `insert` and `next_request`, agreement stage and execution stage use the selector as a producer/consumer queue. The agreement stage submits requests to the selector by calling `insert` in the determined order. Independently, the execution stage fetches the next request to be executed by calling `next_request`, which blocks while there are no requests ready for processing. Please refer to Section 4.2 for a discussion of the use of `force_request` during fault handling.

3.3 Basic Algorithm

The main task of a selector is to determine whether to process a request on its associated replica. For each call to `insert`, the selector executes the algorithm shown in Figure 2 that relies on the following data structures:

- O_{main} the set of currently maintained objects.
- R_{exec} a FIFO queue containing all requests selected for execution. The selector uses this queue to *execute* requests by handing them over to the execution stage on calls to `next_request` (lines 11-12).
- R_{store} a FIFO queue containing all requests not selected for execution.

In general, a selector distinguishes between two categories of requests: First, requests selected for execution on the local replica are forwarded to the execution stage in the order defined by the agreement stage. Second, requests that are not selected for local execution are enqueued in R_{store} preserving their relative order. This approach allows the selector of a non-faulty replica to maintain the following invariant: At any time, the selector is able to bring each object on its local replica up to date. This is the case because either an object already is up to date, as all requests accessing the object have been executed on the local replica, or because the object subsequently can be updated by processing the state-modifying requests from R_{store} that access the object.

When the agreement stage calls `insert`, the selector executes the request analysis function (see Section 2.4) to extract the set of objects O_{req} the request req accesses during execution (line 2). If O_{req} only consists of unmaintained objects, the selector does not select req for execution (line 5). However, if there is *at least one maintained object* in O_{req}, req is selected for execution on the local replica (line 8). For the remainder of Section 3, we assume a request to access only maintained objects. In Section 4, we drop this assumption and also explain the function calls in lines 3 (see Section 4.2.2) and 7 (see Section 4.1).

With every request accessing a maintained object being processed (see Figure 1), local versions of maintained objects are always consistent with the current service state. As we assume that each request accesses at least one state object (see Section 2.4), the algorithm and our state distribution scheme (see Section 3.1) guarantee that, in the absence of faults, each request is processed on at least $f+1$ replicas. In consequence, enough replies are provided to the voter to prove the result correct ($V1$). In case of faults, replies from additional replicas are needed to decide the vote, as further discussed in Section 4.2.1.

3.4 Checkpointing and Garbage Collection

Retaining requests that have not been selected for local execution in R_{store} allows a selector to update each local unmaintained object at any point in time. In order to limit the size of R_{store}, a selector basically uses the same mechanism as [Yin 2003]. It relies on periodic checkpoints that become stable as soon as $f+1$ identical certificates from different replicas are available. When all state changes caused by a request are part of stable checkpoints, a selector is able to discard the request from R_{store}. In contrast to [Yin 2003], our approach involves *object checkpoints* covering only a single state object, instead of full checkpoints covering the whole replica state.

A selector i generates a checkpoint (using the object checkpoint function, see Section 2.3) for an object o for every kth *execution* of a request accessing o, with k being a system-wide constant (e. g., 100); besides object data, the checkpoint also includes digests of the replies to the k requests that led to the checkpoint (see Section 4.2.1). Next, the selector computes a digest d of the object checkpoint and multicasts $\langle CHECKPOINT, o, s, d\rangle_i$ to *all* selectors; s is the sequence number of the request that triggered the checkpoint creation. The selector assembles $f+1$ identical checkpoint messages to a full object-checkpoint certificate that represents the proof for checkpoint correctness.

When the selector completes an object-checkpoint certificate, it discards older checkpoints and checkpoint certificates for the corresponding (maintained or unmaintained) object. A request with sequence number s is deleted from R_{store} as soon as checkpoint certificates (indicating sequence numbers of at least s) of all objects accessed by the request are available.

Using every kth access to an object to decide when to generate an object checkpoint guarantees that all non-faulty replicas assigned to the same object create a consistent checkpoint. As replicas execute all requests accessing a maintained object, they all create the checkpoint after the same request. Therefore, in the absence of faults, at least $f+1$ checkpoint messages are available, enough to assemble a full object-checkpoint certificate. In case of faults, checkpoint messages from additional replicas assist in assembling a full object-checkpoint certificate (see Section 4.2.2).

```
1 void update_objects(Set<Object> O_update, Request req) {
2   Queue<Request> R_selected = ∅;
3   for(int i = R_store.get_index_latest(req); i ≥ 0; −−i) {
4     Request r = R_store.get(i);
5     Set<Object> O_r = RAF(r);
6     if(O_r ∩ O_update ≠ ∅) {
7       R_selected.enqueue(r);
8       O_update = O_update ∪ O_r;
9   } }
10
11  for each Object o in O_update {
12    ObjectCheckpoint ocp = C_store.get(o);
13    if(ocp has not already been applied) {
14      update object using ocp;
15  } }
16
17  for(int i = (R_selected.size() − 1); i ≥ 0; −−i) {
18    Request r = R_selected.get(i);
19    R_store.delete(r);
20    R_exec.enqueue(r);
21 } }
```

Figure 3. Algorithm for updating unmaintained objects.

4. On-Demand Replica Consistency

In this section, we drop the assumptions of all replicas being non-faulty and of requests accessing only maintained objects. As an immediate result of the latter, a selector needs to synchronize the state of its local replica with the current service state before executing a request that accesses unmaintained (and therefore possibly outdated) state objects. However, the selector does not perform a full state update. Instead, the selector ensures *on-demand replica consistency (ODRC)*. ODRC is "on demand" in two dimensions: Consistency is only ensured when a request to be executed actually demands it (time); furthermore, consistency is only ensured for the objects actually accessed by the request (space). In this paper, we use ODRC as a general term for applying selective request execution in conjunction with on-demand replica consistency.

4.1 Handling Cross-Border Requests

As a selector omits requests accessing only unmaintained objects, those objects may become outdated. Therefore, the selector has to ensure consistency of unmaintained objects prior to executing a request that accesses both maintained and unmaintained objects ("*cross-border request*"). The mechanism used by the selector to update unmaintained objects relies on a combination of object checkpoints and additional request execution. To ensure consistency of all objects accessed by a request, the selector calls `update_objects` (see Figure 2, line 7), which executes the two-step algorithm of Figure 3; the algorithm assumes a cache C_{store} holding the latest stable object checkpoints.

```
1 void force_request(Request req) {
2   if(req ∈ R_store) {
3     Set<Object> O_req = RAF(req);
4     update_objects(O_req \ O_main, req);
5     R_exec.enqueue(req);
6     R_store.delete(req);
7 } }
```

Figure 4. Fault-handling function

```
1 void ack_unmain(Set<Object> O_unmain, Request req) {
2   for each Object o in O_unmain {
3     if((++T_access[o] % k) == 0) {
4       attach req to a timer for o;
5       start the timer;
6     }
7 } }
```

Figure 5. Checkpoint monitoring function

In the first step, the selector determines, which requests to execute in order to update all unmaintained objects for a request req. Potential candidates are all requests from R_{store} that up to now were not selected for execution but contribute to updating the state of the unmaintained objects accessed by req. Starting with the latest request in R_{store} whose sequence number is smaller than the sequence number of req, the following operations are repeated for each request r in R_{store}. First, the set of objects O_r accessed by r is composed using the request analysis function (line 5). Second, if any object in O_r is a member in the set of objects to update O_{update}, r contributes to bringing them up to date and is therefore selected for execution (lines 6-7); furthermore, O_{update} is updated adding all objects contained in O_r (line 8), as these objects also have to be consistent when request r will be executed. Note that additional objects in O_r only have to be updated to the extent required by request r, they do not have to contain the current state of the object. In summary, this algorithm step goes back in time selecting all requests accessing objects to update. As these requests may require additional objects to be consistent, those objects are also updated to resemble their state at this point in time.

In the second step, the selector actually restores the replica state. First, it updates each unmaintained object in O_{update} using the checkpoint from C_{store} (lines 11-14). Next, the selector forwards all requests selected in the first step to the execution stage (lines 17-20); $R_{selected}$ is thereby traversed backwards to preserve request order.

Note that `update_objects` is the function that actually provides ODRC: when a request requires a set of objects to be consistent, the selector ensures consistency of exactly those objects, leaving other objects unaffected and possibly outdated. Therefore, executing the request req will produce the same output as if it were processed by a replica whose state is completely up to date.

4.2 Handling Faulty and Slow Replicas

During normal-case operation, a request is executed on $f+1$ replicas as voters only need $f+1$ identical replies to prove a response correct. However, in case of faulty or slow replicas, a voter might not be provided with enough replies to reach a decision. In this case, the voter sends an incomplete-voting notification to all replicas ($V2$).

4.2.1 Handling Incomplete Reply Voting

On the reception of an incomplete-voting notification, the agreement stage performs the standard fault-handling operations (e. g., a view change); in particular, the agreement stage resends the cached reply ($R3$) to the voter. If no reply is available for a request req, the agreement stage forces the selector to select req for local execution using `force_request` (see Figure 4). If req has not yet been processed (i. e., it is in R_{store}; lines 2, 6), the selector treats req like a cross-border request, updates unmaintained objects, and selects req for execution (lines 4-5). As each non-faulty replica eventually learns about an incomplete voting ($V2$), selectors of all non-faulty replicas will eventually select req for execution, providing additional replies to decide the vote.

A replica may receive an incomplete-voting notification for a request whose sequence number is smaller than the sequence number of the latest stable checkpoint of an accessed object. In this case, the selector sends the corresponding reply digest (which is included in the object checkpoint, see Section 3.4) to the voter (omitted in Figure 4). As the object checkpoint (and therefore the reply digest) is stable, there is at least one non-faulty replica that has actually executed the request and therefore resends the full reply on the reception of the incomplete-voting notification. As a result, other non-faulty replicas that have not executed the request may only return a reply digest.

Note that the agreement stage copes with most of the problems arising from faulty or slow replicas, hiding them from the selector. Agreement stages of different replicas decide independently whether to call `force_request`, based on their local reply cache. As voters continue request retransmissions while lacking $f+1$ identical replies, all non-faulty replicas that have originally omitted the execution of the request will eventually provide additional replies.

4.2.2 Handling Incomplete Checkpoint Voting

Faulty or slow replicas may (temporarily) prevent non-faulty replicas from assembling a full object-checkpoint certificate by providing faulty or no checkpoints. In this case, the mechanism in Figure 5 forces other non-faulty replicas to provide additional checkpoints when an object checkpoint does not become stable within a certain period of time. The mechanism relies on a table T_{access} containing an access counter for every unmaintained object and a set of timers.

The basic selector algorithm calls `ack_unmain` for every request req (see Figure 2, line 3); this function increments a counter in T_{access} for each unmaintained object o accessed by req (see Figure 5, lines 2-3). This way, a selector is able to determine the point in time at which the next checkpoint for an unmaintained object is due. A resulting counter value divisible by k indicates that all (non-faulty) replicas maintaining o will checkpoint this object after having executed req (see Section 3.4). In this case, the selector may expect a checkpoint for o to become stable within a certain (application-dependent) period of time. To monitor this, the selector attaches req to an object-specific timer and starts the timer (lines 4-5). The timer is stopped when the selector is able to assemble a full object-checkpoint certificate for o.

When the object timer expires, however, the selector calls `force_request` (see Section 4.2.1) for the request req. As a result, req is executed on the local replica; prior to that, the selector updates the unmaintained object o (see Figure 4, line 4), as o is accessed by req. Furthermore, processing req (i. e., the kth request accessing o since the last stable object checkpoint) triggers the creation and distribution of the next checkpoint for o. As all non-faulty replicas (that do not maintain o) behave in the same manner, each selector is eventually provided with enough additional object checkpoints for o to assemble a full object-checkpoint certificate.

4.3 Safety and Liveness

Introducing a selector between agreement stage and execution stage creates an additional potential point of failure, as a faulty selector may lead to a faulty replica, and vice versa. However, as selector interaction (besides the exchange of object checkpoints) is limited to the local replica (i. e., agreement stage and execution stage), selectors cannot be compromised by other replicas. With our approach treating the agreement stage as a black box, most mechanisms of the surrounding architecture ensuring safety (e. g., committing requests) and liveness (e. g., view changes) remain unaffected.

4.3.1 Safety

The safety properties of ODRC are primarily based on the safety properties of the underlying agreement protocol. As ODRC does not modify the agreement stage, correctness of the agreement protocol is preserved. In particular, it is guaranteed that, in the presence of at most f faults, the totally-ordered sequence of requests provided by the agreement stage is identical on all non-faulty replicas (R1).

In traditional BFT systems, this sequence I_A dictates the order in which all non-faulty replicas execute all requests. Applying ODRC, non-faulty replicas execute requests based on a sequence I_S that is provided by the selector. A correct selector transforms I_A into I_S ensuring the following two properties of I_S: first, requests whose access sets intersect in at least one object appear in I_S in the same relative order as in I_A; second, requests that access different state objects may be reordered. The latter is safe as, for a given initial state, the result of executing those requests in any order places replicas in the same final state. A selector may delay a request accessing only unmaintained objects until the effects of the request are reflected in stable checkpoints for those objects. When a selector has obtained full checkpoint certificates for all objects accessed by a request, it is safe to remove the request from I_S, as at least $f + 1$ replicas have already executed the requests and agreed on the contents of the checkpoints.

A Byzantine selector may provide the selectors of different replicas with different checkpoints for the same object. However, as selectors only apply stable checkpoints, the safety of a non-faulty replica cannot be compromised by a Byzantine selector. Apart from exchanging checkpoints, selectors do not communicate with each other.

Voters (i. e., clients in traditional BFT systems) do not directly interact with ODRC selectors. However, a Byzantine voter may send an incomplete-voting notification for an arbitrary request r to the agreement stage of a non-faulty replica. If r has already been executed on the local replica, the agreement stage resends the cached reply for r (R3). Otherwise, the agreement stage forwards r to the local selector by calling `force_request` (see Section 4.2.1). As the selector ignores all requests that are not included in the totally-ordered sequence of requests provided by the agreement stage, a Byzantine voter is not able to force a selector into executing a request that has never been agreed on. Furthermore, a non-faulty selector will not execute the same request more than once (see Figure 4, lines 2 and 6).

4.3.2 Liveness

In the absence of faults, each request is executed on at least $f + 1$ replicas; that is, voters are able to collect enough identical replies to determine the correct result (V1). In case of network problems or faulty or slow replicas preventing a successful voting, voters eventually inform (e. g., via request retransmission) all selectors of non-faulty replicas about the lack of identical replies (V2). As a result, the affected request will finally be processed on all non-faulty replicas, allowing the client to make progress. Incomplete checkpoint voting is handled similarly, triggered by non-faulty selectors not able to assemble a stable object-checkpoint certificate.

4.4 Throughput Scalability

With all replicas executing all requests, traditional agreement-based BFT systems fail to provide throughput scalability: increasing the number of replicas n (while keeping f, the number of faults to tolerate, constant) does not increase system throughput. To improve throughput scalability, a number of BFT systems have been proposed where only a quorum of replicas handles and processes a request [Abd-El-Malek 2005, Cowling 2006]. However, in order to be safe, any two arbitrary quorums are required to overlap in at least one non-faulty replica. Therefore, quorum size increases with the total number of replicas n.

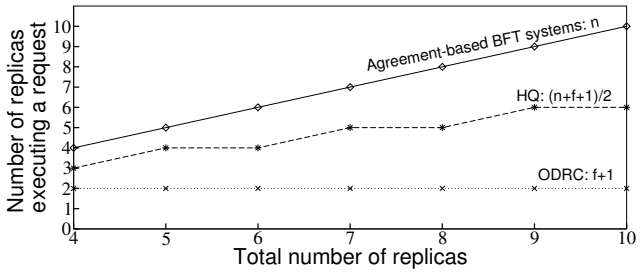


Figure 6. Overview of the number of replicas a request is actually executed on when more replicas are provided than necessary to tolerate $f = 1$ Byzantine fault.

With selectors operating on an agreed totally-ordered sequence of requests, in our approach, there is no need for subsets of replicas executing a request to overlap. Therefore, the minimal subset size of $f + 1$ replicas only depends on the number of faults to tolerate. As shown in Figure 6, for tolerating one Byzantine fault, a request is processed by two replicas, independent of n. In contrast, HQ [Cowling 2006], which requires the smallest quorums, processes a request on $\frac{n+f+1}{2}$ [Merideth 2008] replicas, not scaling as well.

Note that Figure 6 compares the average normal-case operation of a service requiring only few cross-border requests (see Section 8). To improve throughput scalability using ODRC, agreement-based systems need to solve the scalability bottleneck of quadratic costs for agreement [Castro 1999], for example, by using different clusters for agreement and execution [Clement 2009, Yin 2003]. This way, the number of execution replicas can be increased without degrading the performance of the agreement stage. However, our evaluation results show, that ODRC already enables a significant throughput increase for the minimal number of replicas.

5. Optimizations

This section presents optimizations that either directly increase performance of the ODRC algorithms presented in the previous two sections or contribute to the efficiency of the overall system. Furthermore, we describe approaches to configure selectors in order to customize system behavior.

5.1 Dynamic State Distribution

We expect application-specific heuristics to be used to assign objects to replicas; an optimal scheme distributes objects in a way that load is equally balanced across replicas. However, as object access patterns may be subject to change during the lifetime of a service, a static distribution scheme does not guarantee a permanent benefit. To compensate load imbalances, the selector i of an overloaded replica may therefore dynamically delegate the maintenance of an object to another selector j. Note that the fault handling mechanism (see Section 4.2) ensures that other selectors (including i) will step in to tolerate the fault in case j fails to maintain the object after the handover.

5.2 Optimized Checkpointing

We expect the execution stage to make use of copy-on-write and incremental cryptography to reduce the cost of producing object checkpoints [Castro 1999, Rodrigues 2001]. In addition, the following techniques can be applied to optimize checkpoint verification: first, the counter indicating the next checkpoint creation ignores read-only requests not modifying the state; as a result, state objects are only checkpointed when modified. Second, multiple objects may be verified together using a combined certificate.

5.3 Optimistic Updating of Unmaintained Objects

The selector presented in Section 3 updates an unmaintained object when the agreement stage inserts a committed request accessing the object. However, for agreement stages using the standard BFT three-phase protocol, the reception of a pre-prepare message in the first phase of the agreement protocol is already a good indication that a certain request is likely to be committed soon [Kotla 2007]. Therefore, an optimized selector may provide an additional function that allows the agreement stage to hand over a request on the reception of its corresponding pre-prepare message. Based on this information, the selector may optimistically start the updating process for the unmaintained objects accessed by the request in advance. As a result, less updating has to be done when the request is actually committed. In case an announced request is not committed (e. g., due to a faulty primary), this optimization would only lead to unnecessary object updates, it would not compromise safety.

5.4 Proactive Updating of Unmaintained Objects

Selectors may take advantage of periods of reduced workload to proactively update unmaintained objects. As a result, fewer changes have to be reproduced when requests actually demand consistency of unmaintained objects. In order to guarantee an upper bound for ODRC update procedures of unmaintained objects, one might also define a maximum number of outstanding modifying requests, forcing the selector to update an unmaintained object when a certain threshold is reached. In general, there is a tradeoff between reducing replica load and minimizing update duration.

5.5 Optimized Fault Handling

In general, a selector only processes requests exclusively accessing unmaintained objects on calls to `force_request`. Therefore, a crashed replica may result in bad performance, with voters on each request demanding replies from additional replicas due to incomplete voting. An optimized selector may compute statistics on forced requests; on an accumulation of forced requests accessing the same object, the selector may temporarily add the object to its maintained-objects set. This way, voters are provided with additional replies without having to explicitly demand them.

6. NFS Case Study

We have implemented a Byzantine fault-tolerant network file system (NFS) to show the practicality of ODRC. In this section, we discuss how to integrate NFS with ODRC and present an extensive evaluation.

6.1 Service Integration

NFS manages files and directories on the basis of *objects*; we define an ODRC state object to be a single NFS object. Internally, NFS uses *file handles* to uniquely identify an object; there is only one type of file handle, for both files and directories. Most NFS operations access only one object (e. g., `SETATTR`, `GETATTR`, `READ`, `WRITE`). However, some operations access two (e. g., `CREATE`, `REMOVE`, `LOOKUP`) or four (`RENAME`) objects and may therefore lead to cross-border requests.

In general, a request carries the file handle(s) of the object(s) its corresponding operation will read or modify, making it easy to implement a request analysis function. However, there is a set of operations (e. g., `CREATE`, `REMOVE`, `RENAME`) that identify some of the objects accessed by their *name*. Therefore, each selector also maintains a name-to-file-handle mapping for each object. Note that this mapping is only an inconvenience that could be obviated by refactoring the NFS service to either use only file handles or only names to identify objects.

Our prototype comprises parts of the CBASE-FS [Kotla 2004] prototype; CBASE-FS is an extension of the BASE implementation of a Byzantine fault-tolerant network file system [Rodrigues 2001] that supports concurrent execution of requests. In particular, our prototype reuses the CBASE-FS client implementation and the BASE conformance wrapper. Extending the CBASE approach with an ODRC selector module allows us to combine the performance gains offered by processing requests in parallel with the advantages of selective request execution.

As our file system is not the first Byzantine fault-tolerant NFS based on state-machine replication [Castro 1999, Kotla 2007; 2004, Rodrigues 2001, Yin 2003], we omit a discussion of replication-related problems (e. g., non-determinism) that are solved in the BASE abstraction layer. Like BASE, our file system provides NFSv2 [Sun Microsystems 1989].

6.2 Evaluation

We evaluate our file system using a cluster of dual-core hosts (2.4 GHz, 2 GB RAM) for the replicas and a cluster of quad-core hosts (2.4 GHz, 8 GB RAM) for the clients, all hosts are connected with switched 1 Gb/s Ethernet. All experiments use 32 NFS server daemons per replica and a block size of 4 kilobytes for both reads and writes[2]; the data is stored on disk.

[2] 4 kilobytes is the standard block size of CBASE-FS. Note that the maximum block size of NFSv2 is 8 kilobytes. Later versions allow a higher block size and are therefore capable of achieving a higher throughput.

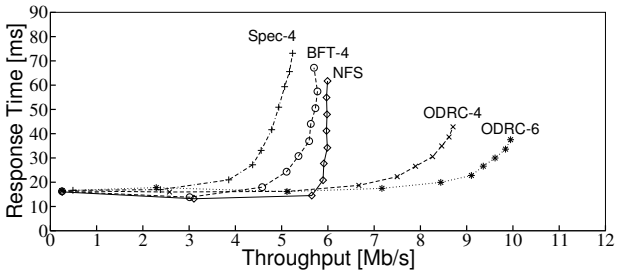


Figure 7. System throughput versus response time at the client for writes of 4 kilobytes with and without ODRC.

The baseline system (*BFT-4*) comprises four replicas and is therefore able to tolerate one Byzantine fault. Each replica hosts an agreement stage and an execution stage that run in separate processes and communicate via local sockets. BFT-4 replicas rely on the three-phase BFT protocol [Castro 1999] to reach agreement. The execution stage of a BFT-4 replica comprises a parallelizer module that allows concurrent execution of requests, as proposed in [Kotla 2004].

We also compare ODRC against a Zyzzyva [Kotla 2007]-like setting (*Spec-4*) that uses speculative execution: all Spec-4 replicas execute a state-modifying request after one protocol phase and send the reply back to the client. The client accepts the reply as soon as $3f + 1$ (read-only requests: $2f + 1$) replicas have provided matching responses. Note that Spec-4 does not implement the full Zyzzyva protocol, only its fast path, which is optimized for the absence of both faults and packet losses.

We evaluate ODRC using two different settings: *ODRC-4* extends the baseline system BFT-4 by introducing a selector component between the agreement stage and the execution stage of each replica; this setting allows us to evaluate the impact of ODRC on response time and throughput for the minimal number of replicas. *ODRC-6* extends ODRC-4 with two additional replicas that only comprise an execution stage but do not participate in the agreement protocol. Instead, they learn the total order of requests from the other replicas [Yin 2003]. ODRC-6 allows us to evaluate the impact of ODRC on scalability. In addition, we run the experiments on unreplicated NFS to get results for the unreplicated case, using the same configuration as for the BFT systems.

6.2.1 Normal-Case Operation

The following experiments evaluate the impact of ODRC on throughput and response time in the absence of faults.

Micro-Benchmark We use a micro-benchmark to evaluate the write throughput and response time of our file system with and without ODRC. In this experiment, we vary the number of clients that continuously write data in blocks of 4 kilobytes to separate files in a directory exported by the file system. In the ODRC test runs, the selector uniformly assigns files to replicas; that is, each ODRC-4 replica treats half (ODRC-6: a third) of all files as maintained objects.

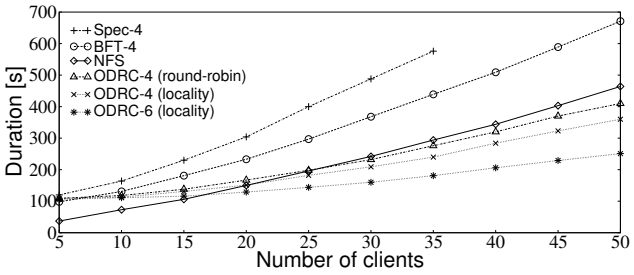
The results of the benchmark (see Figure 7) show that for small workloads, the baseline system BFT-4 achieves slightly better response times than the ODRC variants. This is due to different voting conditions: an ODRC client is able to verify a reply as soon as both (i. e., $f + 1$) replicas processing the request have delivered the correct response. In contrast, a BFT-4 client only needs to wait for the replies of the two fastest non-faulty replicas. This way, the baseline system better compensates delays introduced by the agreement stage and other replication overhead that may vary between replicas. However, our results show that this weakness of ODRC becomes irrelevant for higher workloads.

The baseline system reaches a maximum throughput of 5.7 Mb/s for writes of 4 kilobytes. By selectively executing requests, ODRC-4 reduces the load on replicas and is therefore able to increase the overall throughput by 53%, using the same number of replicas as BFT-4. At the same time, the response time of ODRC-4 is about 30% lower than the response time of BFT-4. Our experiments show that these improvements in throughput and response time even outweigh the overhead of state-machine replication and Byzantine agreement allowing ODRC-4 to outperform unreplicated NFS in both categories. Relying on two additional execution replicas, ODRC-6 is able to increase the throughput to 9.9 Mb/s while further improving response time.

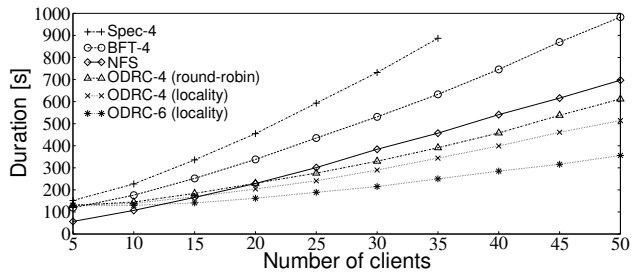
Note that using speculative execution (Spec-4), as for example implemented in Zyzzyva, does not offer any benefits over using traditional BFT in this experiment. The reason for that is that Byzantine agreement only contributes about one to four milliseconds to the response time observed by the clients (more than 14 milliseconds), most of the time is added by the application. Improving agreement through speculative execution therefore has little effect on the overall response time. In contrast, as clients are required to wait for $3f+1$ matching replies instead of only $f+1$, the slowest replica dictates the performance of Spec-4.

Macro-Benchmark We use the Postmark [Katcher 1997] benchmark to evaluate the performance of our Byzantine fault-tolerant file system in a realistic usage scenario. Postmark simulates the usage pattern of modern Internet services such as email or web-based commerce. We apply the same configuration as [Kotla 2004] and run a *read-mostly* experiment where the transaction phase of the benchmark favors reads over writes and a *write-mostly* experiment where reads are dominated by writes. We increase the number of clients (i. e., instances of the Postmark benchmark running in parallel) from five to fifty.

To measure the impact of object distribution on performance, we apply two different strategies to ODRC-4. The first strategy assigns file-system objects (i. e., files and directories) in a *round-robin* fashion to replicas. As this approach completely ignores dependencies between different objects, we consider it a worst-case strategy. The second object distribution strategy uses a simple *locality* heuristic: it assigns a file to the same replicas its parent directory is assigned to; directories are still assigned round-robin and may therefore be assigned to different replicas than their parent directories.



(a) Read-mostly benchmark



(b) Write-mostly benchmark

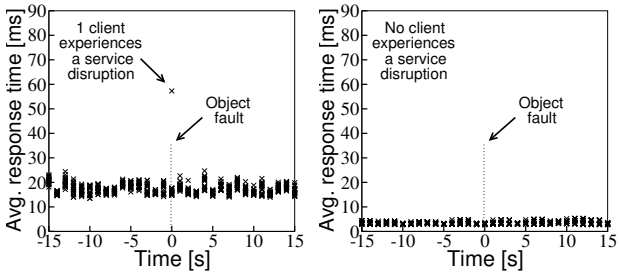
Figure 8. Results of the Postmark benchmark for two scenarios where (a) reads and (b) writes are favored during the transaction phase of the benchmark.

Figure 8 presents the results of the Postmark experiments. For both scenarios, it takes BFT-4 about 50% longer than unreplicated NFS to complete the benchmarks. Applying the round-robin distribution strategy allows ODRC-4 to run the benchmarks in 37% less time than BFT-4 for medium and high workloads. Cross-border requests represent about 11% of all requests for this strategy; assigning files to the same replicas as their parent directory removes almost all of them. Therefore, benchmark durations further decrease for ODRC-4 when using the locality strategy; compared to BFT-4, benchmarks complete in 44% (read-mostly) and 47% (write-mostly) less time. Note that these numbers are close to the theoretical optimum for ODRC-4 of 50% (i. e., a 100% increase in throughput). As a result, ODRC-4 is able to outperform unreplicated NFS by 19% and 25%, respectively.

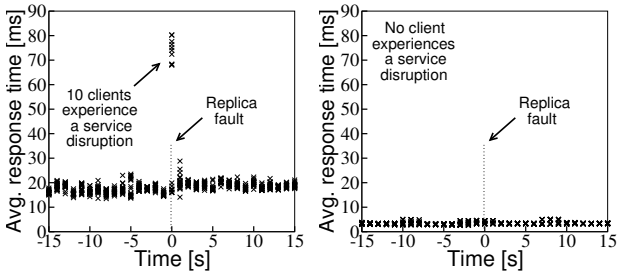
Applying the locality strategy, ODRC-6 completes the benchmarks in about 60% less time than the baseline system; 67% is the optimum for ODRC-6. This confirms the good throughput scalability of the ODRC approach.

6.2.2 Introducing Faults

We now evaluate ODRC in the presence of faults. We distinguish between an *object fault* that leads to corrupted replies on all requests accessing the faulty object, and a *replica fault* that prevents a replica from providing replies at all.

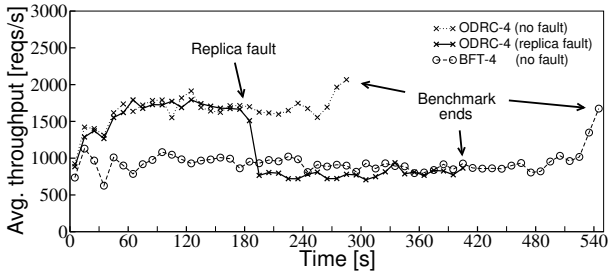


(a) Write benchmark: object fault (b) Read benchmark: object fault

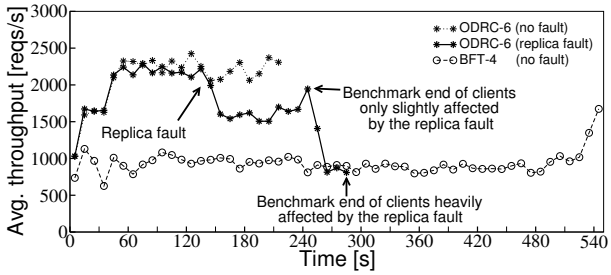


(c) Write benchmark: replica fault (d) Read benchmark: replica fault

Figure 9. Impact of faults on the average response time at the client (1 sample/s) of ODRC-4 for a write-only and a read-only NFS micro-benchmark with twenty clients.



(a) Minimal setting (four replicas)



(b) Extended setting (six replicas)

Figure 10. Impact of a replica fault on the average system throughput of ODRC-4 and ODRC-6 (1 sample / 10 s) for the Postmark write-mostly benchmark with thirty clients.

Micro-Benchmarks This experiment evaluates the impact of faults on ODRC-4 running the NFS write-only micro-benchmark with twenty clients. For comparison, we also run a read-only micro-benchmark where clients read data in blocks of 4 kilobytes from separate files. In each case, we evaluate the worst-case scenario for an object checkpoint interval of $k = 100$ (see Section 3.4); that is, for the write-only benchmark, selectors on non-faulty replicas first have to replay the latest 100 write operations for the affected file(s) before being able to provide a reply to the pending request. For the read-only benchmark, no state modifications need to be replayed[3].

Figure 9a shows that when an object fault occurs during the write-only benchmark, ODRC-4 is not able to provide the client with enough replies to make progress for about one second (resulting in a peak in average response time). During this time, non-faulty replicas that do not maintain the affected file bring their local copies of the file up to date. When this procedure is complete, they process the pending request and provide additional replies. Please refer to Section 8 for a discussion of approaches to minimize the disruption. However, note that none of the clients accessing other files is penalized.

[3] Note that we use an optimization here: as stated in [Kotla 2004], the `READ` operation in NFS modifies the last-accessed-time attribute of a file (i. e., it is a state-modifying operation). However, when replaying multiple successive `READ` requests, only the latest one (i. e., the pending request) actually has to be processed in order to bring the last-accessed-time attribute up to date.

When we inject an object fault during the read-only benchmark, clients observe no notable response time increase (see Figure 9b). With no state modifications to replay, other replicas are prepared to process the pending request right away, tolerating the fault without disruption.

Figures 9c and 9d show that the failure of a replica has a similar impact on system performance than multiple concurrent object faults. In the write-only benchmark, half of clients experience a service disruption as their files are maintained objects on the faulty replica and therefore have to be updated on other replicas. With affected files remaining up to date on non-faulty replicas from then on, there are no disruptions on subsequent requests. However, the average response time increases after a replica fault, as there remain only three non-faulty replicas that process all requests.

Macro-Benchmark This experiment introduces a replica fault during the transaction phase of a write-mostly Postmark benchmark with thirty clients, for both ODRC-4 and ODRC-6. Figure 10 shows the average combined throughput of all clients for this scenario in comparison to BFT-4 and ODRC-4 runs without faults.

When the replica fault occurs, the throughput of ODRC-4 drops by about 50% due to files affected by the fault being restored on non-faulty replicas. Files are updated on the first access after the replica fault has occurred. In consequence, the impact of fault handling is not concentrated to a single point in time but distributed over a period of recovery, allowing ODRC-4 to keep throughput performance above

700 requests/s. With more and more objects being up to date, throughput steadily increases during this phase, eventually reaching the performance level of the baseline system. This behavior is in line with expectations, as in both cases, at this point, all non-faulty replicas process every request.

ODRC-6 comprises more replicas than actually required to tolerate the $f = 1$ Byzantine fault; that is, not all non-faulty replicas need to participate in the handling of a fault. To exploit this, we enable the fault handling for an object on only a set of $3f + 1 = 4$ selectors and disable it on the other two selectors; this is safe regardless of where a fault occurs. Figure 10b shows the benefit of this optimization. When we trigger the replica fault, the throughput of ODRC-6 only decreases by about 30% and does not drop to the BFT-4 level. Furthermore, some clients are able to complete the benchmark in almost the same time as in the absence of faults, as the files they access are maintained on replicas that only play a minor role in the handling of the replica fault; this explains the throughput drop at $t = 255$.

In general, tolerating a replica fault in ODRC-6 is not as costly as in ODRC-4. As objects are distributed across six (instead of four) replicas, a selector in ODRC-6 maintains only a third (instead of half) of all objects. As a result, when a replica fails, other selectors only need to update a third (instead of half) of the service state at most. Increasing the total number of replicas will enhance this advantage.

7. ZooKeeper Case Study

ZooKeeper [Hunt 2010] is a distributed coordination system that provides distributed applications with basic services like leader election, group membership, distributed synchronization, and configuration maintenance. ZooKeeper is widely in use at Yahoo for crucial tasks like failure recovery. Based on the original crash-tolerant implementation, we have built a Byzantine fault-tolerant version of the ZooKeeper service.

7.1 Service Integration

ZooKeeper stores information in a hierarchical namespace. Each tree node is able to store data, and to manage child nodes; we define an ODRC state object to be a single ZooKeeper node. A node is uniquely identified by its path. Each request carries the full path information of the node it will operate on as a string; the ODRC request analysis function uses this string to determine object access.

Our prototype comprises the application logic of the original ZooKeeper implementation. However, in order to make the service Byzantine fault-tolerant, we substitute the crash-tolerant protocol ZooKeeper uses to order requests with the standard three-phase BFT protocol. Furthermore, we introduce voting at the client, and add a small abstraction layer at the replica that ensures consistency of ZooKeeper node metadata (e. g., timestamps and node version counters). Please refer to [Clement 2009] for a discussion of the measures it takes to enforce deterministic ZooKeeper replicas in the context of Byzantine fault tolerance.

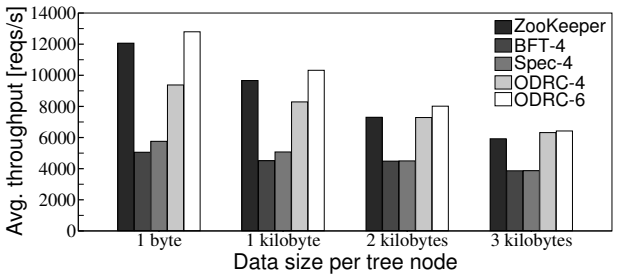


Figure 11. Realized throughput for repeatedly setting the data of ZooKeeper nodes using different data sizes between 1 byte and 3 kilobytes.

7.2 Evaluation

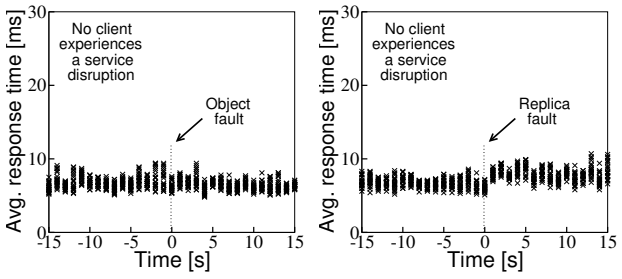
For our evaluation of ZooKeeper, we use the same cluster of machines as well as the same system settings as in Section 6.2. All systems are configured to tolerate one (Byzantine) fault; therefore, the setting for plain crash-tolerant ZooKeeper comprises three replicas.

7.2.1 Normal-Case Operation

We evaluate the write throughput of the different ZooKeeper variants with an experiment in which clients repeatedly write new data chunks of ZooKeeper-typical sizes to nodes; Figure 11 presents the results of this experiment. For one-byte writes, ODRC-4 achieves an 85% improvement over the baseline system BFT-4. As increasing the data size puts more load on the agreement stage, the benefit of ODRC-4 decreases to 63% for writes of 3 kilobytes. For the same reason, the improvement of ODRC-6 over ODRC-4 is higher for small data sizes. However, for large requests, ODRC-4 is able to outperform the crash-tolerant ZooKeeper implementation, using only the minimal number of replicas required for Byzantine fault tolerance. Note that in this experiment, again, speculation (Spec-4) does not provide a substantial performance gain due to the fact that clients have to wait for identical replies from all four replicas to make progress.

7.2.2 Introducing Faults

We repeat the write-only micro-benchmark experiments of Section 6.2.2 for ZooKeeper to examine the worst-case impact of an object fault and a replica fault on the response time of ODRC-4. Figure 12 shows that, in contrast to NFS, faults that occur during the write-only experiment do not lead to a service disruption that is noticeable at the ZooKeeper client. The reason for that lies in the fact that the data of a ZooKeeper node is always written (and also read) atomically and in its entirety; that is, each write operation replaces all the data stored at a node. In consequence, an ODRC selector is able to update the complete data of an unmaintained node by just replaying the latest write request and adjusting the node metadata to reflect the current version number. Exploiting this property of ZooKeeper, which is also provided



(a) Write benchmark: object fault (b) Write benchmark: replica fault

Figure 12. Impact of faults on the average response time (1 sample/s) of ODRC-4 for a write-only ZooKeeper workload from twenty clients.

by other services (e. g., Chubby [Burrows 2006]), allows to significantly speed up fault handling. Of course, the response time also decreases for ZooKeeper after a replica fault (see Figure 12b), as the remaining replicas process all requests.

8. Discussion

Our evaluation shows that ODRC can increase the performance of a replicated BFT system for the default number of replicas by optimizing the resource utilization of execution stages. Thereby, cross-border requests are a key factor limiting ODRC performance gains, as they contradict the goal of executing every request on only a minimal subset of replicas. As the fraction of cross-border requests is not an intrinsic property of a service, but depends to a great extent on the service usage pattern and the object distribution scheme used in ODRC, the occurrence of cross-border requests can be contained by an effective object distribution strategy. For NFS, for example, it suffices to assign files and their parent directories to the same replicas (as done by the locality strategy, see Section 6.2.1) to reduce the fraction of cross-border requests to almost zero. Nevertheless, extensive use of cross-border requests may disqualify a service from ODRC benefits, if refactoring the service is not an option.

Like other BFT systems [Kotla 2007, Wood 2011], ODRC is optimized for fault-free operation. Our evaluation of NFS shows that when a fault occurs, duration of the service disruption depends on how outdated the local version of the affected object is on non-faulty replicas (see Section 6.2.2). In the worst case, the object has to be recreated before being able to make progress. Although this circumstance does not endanger liveness, in practice, additional measures may be taken to trade off some of the performance gains made possible by ODRC in return for a fault-handling speed up. In particular, this can be achieved by proactively updating unmaintained objects (see Section 5.4), limiting the extent to which the local copy of an object becomes outdated. However, as the ZooKeeper example shows, there are also services where such measures are not necessary, as handling a fault does not lead to a service disruption.

The evaluation results for NFS in Section 6.2.1 show that selective request execution improves response times for medium and high workloads. However, ODRC is not capable of reducing the minimal response time of a Byzantine fault-tolerant system, which is mainly dictated by the protocol used for Byzantine agreement. As ODRC uses the agreement stage as a black box and therefore does not rely on a specific protocol for request ordering, the approach can be combined with already available [Clement 2009, Correia 2004, Yin 2003] as well as future systems targeting to reduce the agreement overhead.

9. Related Work

Most related work in the fields of BFT state-machine replication has already been discussed in Section 2, in connection with our system model. In this context, we presented a list of BFT systems [Castro 1999, Correia 2004, Kotla 2004, Reiser 2007, Rodrigues 2001, Yin 2003] that may benefit from ODRC. Zyzzyva [Kotla 2007] and recent work from Guerraoui et al. [Guerraoui 2010] did not make the list as they improve performance by optimistically executing requests in an order that is speculated on. With the request order not being stable (which contradicts R1, see Section 2.2.1), replicas may become inconsistent and have to rely on the help of clients to converge. In contrast, the use of ODRC may lead to partly outdated but never inconsistent replicas. Furthermore, using a selector, a replica is able to update its state without the help of clients and only based on local knowledge.

Yin et al. [Yin 2003] proposed the separation of agreement and execution, which is essential to benefit from the throughput scalability of our approach (see Section 4.4). Clement et al. [Clement 2009] introduced a third stage to reduce authentication cost and optimize request ordering in cluster environments. Prophecy [Sen 2010] improves throughput and latency of BFT state-machine replication for read-centric workloads by introducing a trusted reply-checksum cache. We consider the latter two approaches orthogonal to our work.

Kotla et al. [Kotla 2004] (CBASE) proposed to increase the performance of a BFT system by executing independent requests in parallel. Two requests are independent, if replicas can process the requests in a different order without compromising correctness; this is true, for example, if the requests access different parts of the replica state. In CBASE, information about state access of requests is used by a parallelizer module located between agreement stage and execution stage that forwards concurrent requests to a set of worker threads. Still, all CBASE replicas process all requests. Like CBASE, ODRC uses information about the state access of requests to optimize system performance. However, ODRC replicas only process requests that have been selected for execution by the local selector module. Both approaches can be combined by forwarding the output of a selector to a parallelizer, as implemented in our pro-

totype. Our evaluations for both NFS and ZooKeeper show that combining concurrent execution with selective execution (ODRC-4) allows a significant performance improvement over plain concurrent execution (BFT-4).

Processing full requests on only a subset of replicas was originally proposed by Pâris [Pâris 1986] in the context of a crash-tolerant file system that relies on quorums. The file system distinguishes between replicas that contain the complete file data and a version number ("copies") and replicas that only contain file version numbers but no data ("witnesses"). While copies execute the full request, witnesses only increment their local file version counter when processing a modifying request; still, witnesses participate in each operation. In case of faults, witnesses may be upgraded to copies. Liskov et al. implemented a related approach in the Harp file system [Liskov 1991] that makes active use of witnesses only during times of node failures or network partitions. Cheap Paxos [Lamport 2004] generalizes the idea to use lightweight auxiliary nodes for handling crashes of full-fledged replicas in order to reduce the resource requirements of a replicated service. Ladin et al. [Ladin 1992] have shown that by processing a request on only a subset of replicas and lazily updating the other replicas, a crash-tolerant replicated service can be built that outperforms an unreplicated service.

SPARE [Distler 2011] and ZZ [Wood 2011] are designed to minimize resource consumption in the context of Byzantine fault tolerance. Both systems rely on virtualization and use only $f + 1$ service replicas during normal-case operation. In case of faults or suspected faulty behavior, the systems quickly activate up to f additional replicas running in separate virtual machines. SPARE and ZZ focus on minimizing the resource footprint of a BFT system by reducing the number of service replicas; they were not designed to increase performance. ODRC also proposes an optimization that exploits the fact that $f + 1$ replies are enough to prove a response correct in the absence of faults. However, ODRC uses this property to improve resource utilization of execution stages for the default number of replicas (i. e., $3f + 1$ in traditional BFT systems), which allows to increase the upper throughput bound of the overall system.

State partitioning was previously applied to tolerate crashes and increase scalability in large-scale file systems and distributed data storage [Gribble 2000, MacCormick 2004, van Renesse 2004]. State partitioning was also used in the context of Byzantine fault tolerance: Farsite [Adya 2002] and OceanStore [Rhea 2003] are large-scale file systems that assign files to different groups of $3f + 1$ replicas, each separately executing a BFT protocol. As a result, the throughput of the overall system improves for an increased number of replica groups. However, within a replica group, all replicas execute all requests. Our approach increases throughput by applying state partitioning within a replica group, obviating the need for complex inter–replica-group protocols to handle cross-border requests. Therefore, ODRC already achieves a significant performance improvement for the default number of replicas. However, optimizing the performance of a single replica group, a possible use case of ODRC is to act as a building block for large-scale systems.

Malek et al. [Abd-El-Malek 2005] used quorums to build a BFT system for arbitrary services (Q/U). It requires a minimum of $5f + 1$ servers and a quorum size of $4f + 1$ to tolerate f Byzantine faults. HQ [Cowling 2006] implements a hybrid approach that combines quorum-based and agreement-based protocols to reduce the number of servers to $3f+1$ and the minimal quorum size to $2f + 1$. Both systems make use of quorums to achieve fault scalability. As discussed in Section 4.4, ODRC offers better throughput scalability. In Q/U and HQ, concurrent object access may lead to inconsistent replica states, which requires replicas to revert previous state modifications. In contrast, replica states in our approach may partially become outdated, but never inconsistent.

10. Conclusion

A traditional agreement-based Byzantine fault-tolerant system executes all requests on all replicas to ensure consistency. As a result, the system usually provides more than the $f+1$ identical replies to a request that are actually needed for a client to make progress in the absence of faults.

In this paper, we have shown that a selector module between agreement stage and execution stage is able to increase the performance of the overall BFT system by selectively executing each request on only a subset of $f + 1$ replicas, based on the state objects a request accesses. As this approach may lead to parts of the replica state being outdated, the selector ensures replica consistency by updating the state of objects on demand, that is, at the time and to the extent required by a request to be executed.

Our evaluation of Byzantine fault-tolerant variants of NFS and ZooKeeper shows that the use of on-demand replica consistency lowers the response time for medium and high workloads, and is able to increase the throughput of a Byzantine fault-tolerant service beyond the throughput of the corresponding non–fault-tolerant unreplicated service.

Acknowledgements

We would like to thank the anonymous reviewers for their comments and our shepherd, Liuba Shrira, for her guidance.

References

[Abd-El-Malek 2005] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie. Fault-scalable Byzantine fault-tolerant services. In *Proceedings of the 20th Symposium on Operating Systems Principles*, pages 59–74, 2005.

[Adya 2002] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, pages 1–14, 2002.

[Burrows 2006] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, pages 335–350, 2006.

[Castro 1999] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, pages 173–186, 1999.

[Chun 2007] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz. Attested append-only memory: making adversaries stick to their word. In *Proceedings of the 21st Symposium on Operating Systems Principles*, pages 189–204, 2007.

[Clement 2009] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riche. UpRight cluster services. In *Proceedings of the 22nd Symposium on Operating Systems Principles*, pages 277–290, 2009.

[Correia 2004] M. Correia, N. F. Neves, and P. Veríssimo. How to tolerate half less one Byzantine nodes in practical distributed systems. In *Proceedings of the 23rd International Symposium on Reliable Distributed Systems*, pages 174–183, 2004.

[Cowling 2006] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira. HQ replication: A hybrid quorum protocol for Byzantine fault tolerance. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, pages 177–190, 2006.

[Distler 2011] T. Distler, R. Kapitza, I. Popov, H. P. Reiser, and W. Schröder-Preikschat. SPARE: Replicas on hold. In *Proceedings of the 18th Network and Distributed System Security Symposium*, 2011.

[Gribble 2000] S. D. Gribble, E. A. Brewer, J. M. Hellerstein, and D. Culler. Scalable, distributed data structures for Internet service construction. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation*, pages 319–332, 2000.

[Guerraoui 2010] R. Guerraoui, N. Knežević, V. Quéma, and M. Vukolić. The next 700 BFT protocols. In *Proceedings of the EuroSys 2010 Conference*, pages 363–376, 2010.

[Hendricks 2007] J. Hendricks, G. R. Ganger, and M. K. Reiter. Low-overhead Byzantine fault-tolerant storage. In *Proceedings of the 21st Symposium on Operating Systems Principles*, pages 73–86, 2007.

[Hunt 2010] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Annual Technical Conference*, pages 145–158, 2010.

[Katcher 1997] J. Katcher. Postmark: A new file system benchmark. Technical Report 3022, Network Appliance Inc., 1997.

[Kotla 2007] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: speculative Byzantine fault tolerance. In *Proceedings of the 21st Symposium on Operating Systems Principles*, pages 45–58, 2007.

[Kotla 2004] R. Kotla and M. Dahlin. High throughput Byzantine fault tolerance. In *Proceedings of the 2004 Conference on Dependable Systems and Networks*, pages 575–584, 2004.

[Ladin 1992] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Providing high availability using lazy replication. *ACM Transactions on Computer Systems*, 10:360–391, 1992.

[Lamport 2004] L. Lamport and M. Massa. Cheap Paxos. In *Proceedings of the 2004 International Conference on Dependable Systems and Networks*, pages 307–314, 2004.

[Liskov 1991] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, L. Shrira, and M. Williams. Replication in the Harp file system. In *Proceedings of the 13th Symposium on Operating Systems Principles*, pages 226–238, 1991.

[MacCormick 2004] J. MacCormick, N. Murphy, M. Najork, C. A. Thekkath, and L. Zhou. Boxwood: abstractions as the foundation for storage infrastructure. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, pages 105–120, 2004.

[Merideth 2008] M. Merideth. Tradeoffs in Byzantine-fault-tolerant state-machine-replication protocol design. Technical Report CMU-ISR-08-110, Carnegie Mellon University, 2008.

[Pâris 1986] J.-F. Pâris. Voting with witnesses: A consistency scheme for replicated files. In *Proceedings of the 6th International Conference on Distributed Computer Systems*, pages 606–612, 1986.

[Reiser 2007] H. P. Reiser and R. Kapitza. Hypervisor-based efficient proactive recovery. In *Proceedings of the 26th International Symposium on Reliable Distributed Systems*, pages 83–92, 2007.

[Rhea 2003] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz. Pond: The OceanStore prototype. In *Proceedings of the 2nd Conference on File and Storage Technologies*, pages 1–14, 2003.

[Rodrigues 2001] R. Rodrigues, M. Castro, and B. Liskov. BASE: Using abstraction to improve fault tolerance. In *Proceedings of the 18th Symposium on Operating Systems Principles*, pages 15–28, 2001.

[Schneider 1990] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Survey*, 22(4):299–319, 1990.

[Sen 2010] S. Sen, W. Lloyd, and M. J. Freedman. Prophecy: Using history for high-throughput fault tolerance. In *Proceedings of the 7th Symposium on Networked Systems Design and Implementation*, 2010.

[Sun Microsystems 1989] Sun Microsystems. NFS: Network file system protocol specification. Internet RFC 1094, 1989.

[van Renesse 2004] R. van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, pages 91–104, 2004.

[Wester 2009] B. Wester, J. Cowling, E. B. Nightingale, P. M. Chen, J. Flinn, and B. Liskov. Tolerating latency in replicated state machines through client speculation. In *Proceedings of the 6th Symposium on Networked Systems Design and Implementation*, pages 245–260, 2009.

[Wood 2011] T. Wood, R. Singh, A. Venkataramani, P. Shenoy, and E. Cecchet. ZZ and the art of practical BFT execution. In *Proceedings of the EuroSys 2011 Conference*, 2011.

[Yin 2003] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating agreement from execution for Byzantine fault tolerant services. In *Proceedings of the 19th Symposium on Operating Systems Principles*, pages 253–267, 2003.