Scalable Byzantine Fault-tolerant State-Machine Replication on Heterogeneous Servers

Michael Eischer · **Tobias Distler**

Received: 21 September 2017 / Accepted: 11 August 2018

Abstract When provided with more powerful or extra hardware, state-of-the-art Byzantine fault-tolerant (BFT) replication protocols are unable to effectively exploit the additional computing resources: On the one hand, in settings with heterogeneous servers existing protocols cannot fully utilize servers with higher performance capabilities. On the other hand, using more servers than the minimum number of replicas required for Byzantine fault tolerance in general does not lead to improved throughput and latency, but instead actually degrades performance. In this paper, we address these problems with OMADA, a BFT system architecture that is able to benefit from additional hardware resources. To achieve this property while still providing strong consistency, OMADA first parallelizes agreement into multiple groups and then executes the requests handled by different groups in a deterministic order. By varying the number of requests to be ordered between groups as well as the number of groups that a replica participates in between servers, OMADA offers the possibility to individually adjust the resource usage per server. Moreover, the fact that not all replicas need to take part in every group enables the architecture to exploit additional servers.

Keywords Byzantine fault tolerance · State-machine replication · Scalability · Heterogeneity · Resource efficiency

1 Introduction

Applying the concept of Byzantine fault-tolerant (BFT) state-machine replication [8], it is possible to build reliable systems that continue to correctly provide their services even if some of their replicas fail in arbitrary ways. This includes failure scenarios caused by hardware problems as well as (potentially malicious) misbehavior of software components. In order to guarantee consistency in such systems, replicas execute client requests only after the requests have been committed by a BFT agreement

Michael Eischer and Tobias Distler
Friedrich-Alexander University Erlangen-Nürnberg (FAU)
E-mail: {eischer,distler}@cs.fau.de

protocol, which is responsible for establishing a global total order on all requests. In particular, the agreement protocol ensures that the determined order of client requests remains stable in the presence of replica failures or network problems.

In general, BFT agreement protocols require a minimum of $3f+1$ replicas to tolerate up to f faulty replicas [22]. Although the number of protocol participants can be larger than $3f+1$, many BFT systems opt for exactly this many replicas [4, 6, 7, 8, 9, 18, 23]. This is mainly due to the fact that the internal architectures of most state-of-the-art BFT agreement protocols do not allow them to exploit additional replicas. In contrast, with all replicas participating in the ordering of all requests, additional replicas usually come at the cost of an increased computational and network overhead, and consequently degrade performance without offering any notable advantages.

To prevent the agreement protocol from becoming the bottleneck of the entire BFT system, research efforts in recent years aimed at increasing the throughput of BFT agreement while keeping the number of replicas at a minimum [6, 18]. However, these approaches are based on the assumption that all replicas run on homogeneous servers, that is, servers with equal or at least similar performance capabilities. Unfortunately, it is not always possible to operate a BFT system under such conditions. Especially in cloud deployments, the performance capabilities of different virtual machines can vary significantly even if they are of the same instance type [20]. This is usually a consequence of virtual machines being run on heterogeneous physical servers, making it very difficult for cloud providers to offer identical computing resources across virtual machines. As a result, it is basically impossible to ensure the homogeneity of virtualized servers when deploying a BFT system in the cloud.

To address the problems discussed above, we present OMADA, a BFT system architecture that exploits computing resources existing approaches are not able to utilize: additional agreement replicas as well as spare capacities on fast servers. OMADA achieves this by parallelizing agreement into multiple heterogeneous groups and varying the ordering workload between them. This allows OMADA to individually adjust the responsibilities of replicas to the particular performance capabilities of their servers. For example, a replica on a more powerful server can participate in more than one group and be responsible for ordering a large fraction of requests, whereas a replica on a less powerful server might only be part of a single group.

Although in this paper we primarily focus on heterogeneity introduced by servers, we expect our approach to also be beneficial for scenarios in which variations between replicas are the result of other sources of heterogeneity. For example, using heterogeneous replica implementations in order to minimize the probability of common mode failures [9, 11, 15] in general also causes replicas to advance at different speeds, consequently having a similar effect as servers with different capabilities.

In summary, this paper makes four contributions: (1) It presents OMADA, a BFT system architecture that benefits from additional replicas. (2) It details how OMADA can exploit servers with heterogeneous performance capabilities. (3) It shows that OMADA is generic by integrating the architecture with two BFT agreement protocols. (4) It evaluates OMADA in a heterogeneous setting. In the remainder of the paper, Sect. 2 identifies limitations of existing BFT architectures, Sect. 3 presents our approach to address these issues, Sect. 4 describes two OMADA implementations, Sect. 5 evaluates OMADA, Sect. 6 discusses related work, and Sect. 7 concludes.

2 Background and Problem Statement

In this section, we first give an overview of BFT systems and then analyze the scalability and resource usage of state-of-the-art BFT agreement protocols.

2.1 Background

In general, BFT systems based on state-machine replication [4, 6, 7, 8, 9, 18, 23] require $n \geq 3f+1$ replicas to tolerate up to f faulty replicas. As shown in Fig. 1, replicas ensure consistency by first running a protocol to agree on a client request before executing the request. For this purpose, one of the replicas in the system acts as *leader* while all other replicas participate as *followers*. If the leader becomes faulty, replicas initiate a *view change* to reassign the leader role to a different replica.

Having received a request from a client, the leader assigns a unique sequence number to the request and then starts the agreement process consisting of two rounds of all-to-all communication between replicas: In the first round, which consists of two phases (i.e., *pre-prepare* and *prepare*), replicas ensure that they consider the same request proposal by the leader. After that, the second round (i.e., the *commit* phase) is responsible for finalizing the assignment of the sequence number to the particular request. In both cases, a replica completes a round once it has collected a quorum of size $\lceil \frac{n+f+1}{2} \rceil$ of matching messages. The quorum size guarantees that each possible pair of quorums intersects in at least $f+1$ arbitrary replicas, and therefore in at least one correct replica. Requests for which the agreement process has completed on a replica are executed in the order of their sequence numbers. A client accepts the result to its request after having obtained $f+1$ matching replies from different replicas as this guarantees that at least one of the replies was sent by a correct replica.

In order to prevent a faulty replica from impersonating a correct replica, correct replicas authenticate each message, usually using a MAC authenticator, that is, a vector of message authentication codes [8]. Each MAC in the vector is calculated using a secret only known to the sender and a particular receiver and cannot be verified by a third party, thus requiring an authenticator to contain an individual MAC for each intended recipient of the message. As a result, both the size of a MAC authenticator and the computational cost of creating it are proportional to the number of recipients.

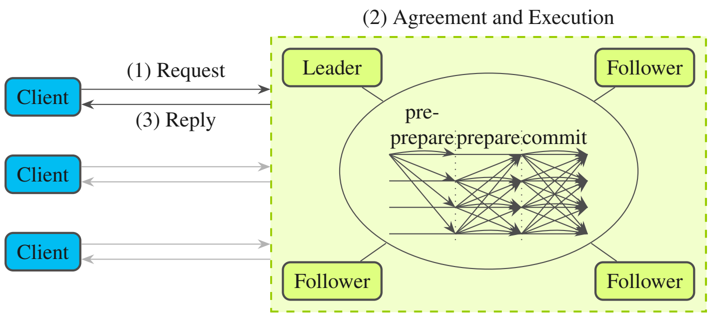


Fig. 1 Overview of a BFT system: Requests are first ordered and then executed by each non-faulty replica.

2.2 Problem Statement

Building on the basic approach presented in Sect. 2.1, in recent years different works have proposed architectural changes and protocol refinements, for example, to improve resilience [4] or reduce replication costs [12, 16, 25]. Below, we focus on two problems that so far remain unsolved: The ability of a BFT system to scale with the number of agreement replicas as well as the efficient use of heterogeneous servers.

Lack of Scalability For applications for which the computational cost of executing a client request is comparably small (e.g., coordination services [14]), the agreement stage of a BFT system usually is the decisive factor limiting performance. Unfortunately, introducing additional agreement replicas to solve this issue for two reasons is not an option in existing systems: First, due to the fact that as discussed in Sect. 2.1 the quorum size depends on the total number of replicas, adding replicas leads to larger quorums and consequently requires more messages. Second, when the number of intended recipients increases, creating MAC authenticators for the messages exchanged between replicas becomes more costly and the messages become larger.

Inefficient Use of Heterogeneous Servers With all replicas participating in both the agreement and the execution of all requests, the replicas in a BFT system usually consume a similar amount of processing resources. Some protocols even deliberately minimize potential imbalances caused by the additional responsibilities of a leader by rotating the leader role among replicas [6, 23, 24]. While a balanced resource usage is beneficial if replicas run on servers that have the same performance capabilities, it prevents existing BFT systems from fully utilizing the available resources if replicas are executed on heterogeneous servers. Due to progress depending on a quorum of replicas, in such environments the performance of the agreement stage is limited by the $\lceil \frac{n+f+1}{2} \rceil$th fastest server, leaving resources on more powerful machines unused.

Summary To be able to benefit from additional agreement replicas, a BFT system must ensure consistency without involving all replicas in all message exchanges. Furthermore, to exploit heterogeneous servers such a system must provide means to distribute load depending on the specific performance capabilities of each server.

3 OMADA

In this section, we present details of the OMADA system architecture and explain how it is able to exploit additional replicas as well as spare capacities on servers with heterogeneous performance capabilities. As illustrated in Fig. 2, to use additional servers OMADA parallelizes the agreement of client requests into multiple groups and in addition also separates agreement from execution [25]; that is, client requests not necessarily need to be processed by the same replicas by which they have been ordered. As a consequence, replicas in OMADA may assume different roles that are associated with different responsibilities: ordering client requests by participating in an agreement group (*leader* or *follower*) and executing client requests (*executor*).

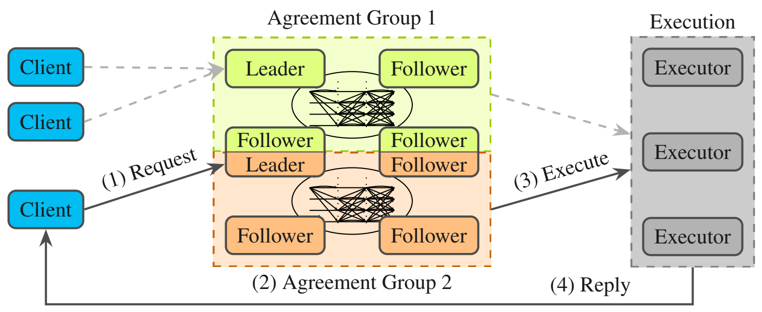


Fig. 2 Overview of the OMADA system architecture relying on multiple, possibly overlapping, agreement groups. To invoke an operation at the application, a client (1) sends a request to one of the groups, which then (2) orders the request using a BFT agreement protocol and eventually (3) forwards the request to a set of executors. Having processed the request, (4) the executors return their results to the client.

To support heterogeneous servers, a replica in OMADA can participate in more than one agreement group and furthermore assume multiple roles. This approach allows OMADA to tailor the responsibilities of each replica to the individual performance capabilities of its server. While a replica on a powerful server, for example, may be part of several agreement groups and also act as executor, a replica on a slow server might only contribute to request ordering in a single agreement group.

Despite relying on multiple, largely independent agreement groups, OMADA is nevertheless able to establish a total order on all requests. To achieve this, OMADA splits the sequence-number space into partitions of equal size and statically maps one partition to each group. In particular a group g is responsible for assigning the sequence numbers $S_g = \{k \cdot |\mathscr{G}| + g | k \in \mathbf{N}\}$; $\mathscr{G}$ denotes the set of all groups. This approach of parallelizing agreement into multiple groups has the key advantage that the messages required for ordering a request only need to be exchanged between replicas of the respective group, not between all agreement replicas in the entire system.

Knowledge about the number, composition, and individual sequence-number partitions of agreement groups, as well as the information which replicas act as executors, is static and available throughout the system. This, for example, allows a client to randomly select an agreement group at start up, which from then on will be responsible for handling all of the subsequent requests the client issues to the service.

3.1 Scalable Ordering Based on Multiple Agreement Groups

In the following, we present the overall protocol OMADA runs to ensure that requests are ordered and executed properly. As the OMADA system architecture does not depend on a specific agreement method, we also define the requirements an agreement protocol needs to fulfill in order to be used within an agreement group. Finally, we discuss specifics of OMADA such as the coordination of groups and fault handling. We use $\langle m \rangle \alpha_{i,\mathscr{R}}$ to denote a message m that has multiple recipients and is therefore authenticated with a MAC vector containing MACs between the sender i and each recipient j in the set $\mathscr{R}$. Besides, $\langle m \rangle \mu_{i,j}$ represents a message that is exchanged between sender i and a single recipient j and authenticated with a single MAC.

Overall Protocol To access the application, a client c sends a $\langle \text{REQUEST}, c, o, t \rangle \alpha_{c,\mathscr{A}}$ message to its agreement group. As the request will only be verified by members of this group, the authenticator of this message is limited to MACs for the group's agreement replicas $\mathscr{A}$. Apart from the command to execute o, the request also contains a client-local timestamp t that is incremented by the client on each operation. As agreement replicas store the timestamp t_c of the latest committed request of each client, the timestamp t allows them to detect and consequently ignore old requests.

Having received the request, the agreement group first verifies that the message is authentic and then starts the agreement process (see below) to assign a unique sequence number s to the request; s is chosen as the lowest of the agreement group's unused sequence numbers. Once the request is committed, each agreement replica a sends an $\langle \text{EXECUTE}, s, q, a, cri \rangle \alpha_{a,\mathscr{E}}$ message to all executors $\mathscr{E}$. Apart from the client request q, this message also comprises a field *cri* containing information that later enables a client to determine which agreement replica it should contact for a subsequent request. As discussed in Sect. 4, the exact content of the *cri* field depends on the specific agreement protocol used to order requests within the agreement group.

To tolerate up to f faults, OMADA relies on a total of $2f+1$ executors. An executor only accepts an EXECUTE if the message is authentic and its sender i is indeed an agreement replica of the group responsible for assigning sequence number s. Before executing the corresponding request, an executor first waits until having obtained $f+1$ matching EXECUTEs from different agreement replicas, as this proves that at least one correct replica has committed the request. As the same request may be committed at different times on different replicas, correct agreement replicas not necessarily provide the same contact-replica information in their respective EXECUTE. Therefore, an executor ignores the *cri* field when comparing these messages.

Although EXECUTEs potentially arrive in a nondeterministic pattern, executors process client requests in the order of their sequence numbers, leaving no gaps between sequence numbers. Similar to agreement replicas, executors manage a timestamp t_c for each client, which is the timestamp of the latest request of a client c the executor has processed. To prevent multiple invocations of the same request, the execution of a request from a client c with a timestamp $t \leq t_c$ consists of a no-op.

After an executor e has processed a request, the executor sends the result r in a $\langle \text{REPLY}, c, t, e, r, \vec{cri} \rangle \mu_{e,c}$ message to the client c; $\vec{cri}$ is a vector of the contact-replica information contained in the $f+1$ corresponding EXECUTEs (see Sect. 4). A client accepts a result after having received $f+1$ REPLYs with matching r from different executors as this guarantees that at least one of the messages originates from a correct executor and therefore contains the correct result. As the contact replica of a group might change over time, the client ignores the $\vec{cri}$ vector when comparing replies.

Internal Agreement-Group Protocol OMADA's overall protocol presented above does not specify how client requests are ordered within an agreement group. As a consequence, it is possible to integrate the OMADA system architecture with different agreement protocols. In the following, we present the general requirements an agreement protocol needs to fulfill in order to be used in OMADA; please refer to Sect. 4 for a discussion of two concrete prototype implementations.

```
/* Agreement on client requests */
void orderRequest(REQUEST request);
[REQUEST, SEQUENCENUMBER, CONTACTREPLICAINFORMATION] getCommittedRequest();

/* Skipping sequence numbers */
void flush(SEQUENCENUMBER seqNr);

/* Garbage collection */
void discard(SEQUENCENUMBER seqNr);
```

Fig. 3 Interface used by OMADA to access the group-internal agreement protocol (pseudo code).

- *Byzantine fault tolerance*: Being a BFT system architecture, OMADA requires a replication protocol to provide safety in the presence of up to f arbitrary replica failures. In this context, OMADA does not pose any restrictions on how many agreement replicas a protocol must rely on for this purpose, although to simplify presentation we assume that an agreement group consists of $3f+1$ replicas, because this is the most common group size for BFT protocols based on state-machine replication [3, 4, 6, 7, 8, 10, 23]. With regard to synchrony, OMADA introduces no additional assumptions. Consequently, the synchrony model of the overall system depends on the synchrony model of the underlying BFT protocol.
- *Stable total order*: To be compatible with OMADA, an agreement protocol must produce a sequence of totally-ordered client requests in which, once a request is committed, the assignment of a sequence number to the request does not change anymore. As a key benefit, this requirement greatly simplifies the interaction between agreement and execution stage, and in addition also frees executors in OMADA from the need to have a rollback mechanism for the application state. Note that the necessity of having to establish a stable order still allows an agreement protocol to deliver committed requests out of order [8] as long as each request is unchangeably mapped to a unique sequence number.

As a consequence of OMADA only imposing very generic assumptions on the internal agreement-group protocol, a variety of state-of-the-art BFT protocols can be used to order requests (e.g., [3, 4, 6, 7, 8, 10, 23]). Furthermore, the integration of an existing protocol into OMADA is facilitated by the lightweight interface handling the interaction between the architecture and the agreement protocol, which is illustrated in Fig. 3. In particular, this interface includes methods with which the architecture can start the agreement process for new client requests (`orderRequest()`) and collect requests for which agreement has completed (`getCommittedRequest()`). Besides, there are methods for instructing the agreement protocol to skip sequence numbers (`flush()`) and performing garbage collection of agreement-protocol messages (`discard()`), whose rationale and use cases are further explained below.

Coordination of Agreement Groups Agreement groups in OMADA operate independently of each other and therefore possibly advance at different speeds. As a result, one group may for example already have committed a client request for sequence number s while another group has not yet reached sequence number $s-1$. To ensure liveness in such scenarios, OMADA provides a mechanism that allows slow agreement groups to detect that they have fallen behind by receiving notifications from executors when requests with higher sequence numbers become ready for processing.

To detect gaps in the sequence of executable requests, each executor in OMADA maintains information about s_{exec}, the sequence number of the last client request it has executed, and s, the highest sequence number for which the executor has collected $f+1$ matching EXECUTEs. Whenever one of these values changes, an executor e broadcasts a $\langle \text{FLUSH}, s_{exec}, s, e \rangle \alpha_{e,\mathscr{X}}$ message to all agreement replicas $\mathscr{X}$; in addition, to ensure that the agreement replicas eventually receive the information, the executor periodically rebroadcasts the latest FLUSH message with the current values. By combining the information contained in FLUSH messages from different executors, agreement replicas are able to reliably determine the overall system progress. For this purpose, each agreement replica calculates $s_{progress}$ to be the $f+1$ highest sequence number s the replica has learned from different executors.

Based on a comparison of $s_{progress}$ with the latest sequence number s_g for which the agreement process has been started, replicas of a group g can determine whether their group has fallen behind in relation to other groups. If this is the case, the group uses the agreement-protocol interface's `flush()` method (see Fig. 3) to start new protocol instances for all of the group's sequence numbers between s_g and $s_{progress}$, proposing either a client request (if available) or a no-op. Consequently, the sequence-number gap that temporarily prevents executors from processing further requests will eventually be closed, enabling the system to make progress again.

To ensure liveness in the presence of faults, agreement replicas monitor the behavior of their group and initiate the necessary fault-handling procedures (e.g., by triggering a view change) if the protocol instances that are required to close the gap are not started within a certain predefined period of time. Apart from that, to tolerate message losses agreement replicas retransmit EXECUTEs for sequence numbers higher than s_{exec} when receiving repeated FLUSH messages from an executor.

Executor Checkpoints With EXECUTEs not necessarily arriving in the order of their sequence numbers, executors may need to buffer them. To implement a bounded buffer, an executor uses a sliding window of size $W = 2 * cp_{interval}$ [10] and only stores EXECUTEs with numbers between s_{low} and $s_{low} + W$. To advance the window, in intervals of $cp_{interval}$ each executor e creates and stores a checkpoint cp of the application state, the latest client timestamps, and the latest reply it has sent to each client. Furthermore, the executor broadcasts a $\langle \text{CHECKPOINT}, s, D(cp), e \rangle \alpha_{e,\mathscr{X} \cup \mathscr{E}}$ message to all agreement replicas and executors; s is the sequence number of the latest request processed prior to the snapshot and $D(cp)$ denotes a hash of the checkpoint.

When an executor receives $f+1$ matching CHECKPOINTs from different executors for a sequence number $s > s_{low}$, the checkpoint becomes stable. At this point, the executor sets the start of its local window to sequence number s and discards all EXECUTEs and checkpoints before $s - cp_{interval}$. If an executor has fallen behind, advancing the window can result in requests being skipped. To ensure a consistent state in such scenarios, an executor first obtains a full checkpoint with matching sequence number and hash from another executor before continuing to process further requests.

Besides guaranteeing execution-stage progress, CHECKPOINTs also enable agreement groups to perform garbage collection of internal messages. For this purpose, an agreement replica notifies its local agreement protocol about stable checkpoints by invoking the protocol interface's `discard()` method (see Fig. 3).

Fault Handling OMADA tolerates up to f faulty replicas per agreement group and a maximum of f faulty executors. In heterogeneous settings where some replicas assume multiple roles, the failure of a replica can affect more than one component. Relying on a Byzantine fault-tolerant protocol for request ordering within each group has the key advantage that for many fault scenarios, OMADA does not need to provide additional mechanisms, as they are already handled by the agreement protocol.

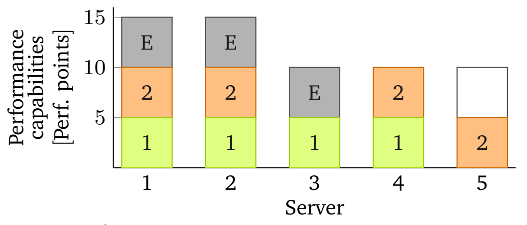
If a client issues a request but does not get a result within a predefined, BFT-protocol-specific period of time, the client sends the request to both all replicas of its agreement group as well as all executors. This way, replicas learn about the problem and if necessary can initiate fault handling within their group or retransmit the EXECUTE for a committed request to handle cases in which previous messages to executors have been lost due to network problems. On the other hand, executors resend the corresponding reply (if available) when receiving a request directly from a client.

Optimizations OMADA supports common BFT-system optimizations such as payload hashes and batching, that is, ordering multiple requests in the same agreement instance [8]. If batching is applied, clients use the individual maximum batch sizes of agreement groups (see Sect. 3.2) as relative weights when randomly selecting a group. To improve efficiency, the leader of a group g aims at proposing batches of maximum size $b_{g,max}$ as long as this does not introduce unnecessary delays. In particular, this means that in cases in which there are less than $b_{g,max}$ new requests, the leader only proposes the batch if, based on the progress information contained in the executors' latest FLUSH messages, the leader knows that its group has fallen behind and that therefore its batch will be immediately processed when arriving at the execution stage. In contrast, if the FLUSHes indicate that its group is ahead of other groups, the leader defers the proposal of the batch in favor of waiting for additional requests.

For replicas assuming more than one role, OMADA offers the following optimizations: First, messages to multiple receivers such as FLUSHes need to be sent only once to each server. Second, if the same replica acts both as an agreement replica as well as an executor, a request becomes ready for processing as soon as it has been committed locally; the executor does not have to wait for an external proof in the form of $f+1$ matching EXECUTEs. As a result, it is sufficient for agreement replicas to only send EXECUTEs to those executors whose replicas are not part of the same group.

3.2 Supporting Heterogeneous Servers

To effectively exploit the resources available in heterogeneous settings, OMADA statically tailors the responsibilities of each replica to the individual performance capabilities of its server before startup. In the following, we describe the systematic approach to determine the assignment of roles to replicas we use for this purpose: First, we assess the specific performance capabilities of each server in the system. Next, we estimate how many resources to reserve for the agreement stage compared with the execution stage. Then, we rely on an integer linear program to determine the number of agreement groups as well as the mapping of roles to replicas. Finally, in the last step we define an individual maximum batch size for each agreement group.



Variable	Description
$\mathcal{R}$	Set of all servers
$\mathcal{E}$	Set of executors (i.e., the 2f+1 servers with the most performance points in the system)
l_r	Performance points of server r
G_{max}	Maximum number of groups
$\mathcal{P}$	All possible group permutations
x_g	Performance points for group $g \in \mathcal{P}$
o_g	Number of executors without a collocated agreement replica of group g

Group Assignment

```
Variables
$w = \max_{r \in \mathcal{R}}(l_r)$
$cf_g = \frac{1}{1+o_g * ci_a}$
$\mathcal{P} = \{p | p \subset \mathcal{R} \wedge |p| = 3f+1\}$
$|\mathcal{P}| = \binom{|\mathcal{R}|}{3f+1}$

Constraints
$\forall r \in \mathcal{R}, r \notin \mathcal{E}:$
        $\sum_{g \in \mathcal{P}, r \in g} x_g \leq l_r$
$\forall r \in \mathcal{E}: x_e$
        $+ \sum_{g \in \mathcal{P}, r \in g} x_g \leq l_r$
$\sum_{g \in \mathcal{P}}(0.5 x_g) - 0.5 \leq x_e$

$\forall g \in \mathcal{P}: x_g + w * x_{gp} \leq w$
$\sum_{g \in \mathcal{P}} x_{gp} \geq |\mathcal{P}| - G_{max}$

Optimization target
max($\sum_{g \in \mathcal{P}} cf_g * x_g$)
```

Fig. 4 Integer linear program for systematically assigning roles to replicas in OMADA. To account for efficiency gains achieved by collocating an agreement replica with an executor, the program weights agreement groups using a cost factor cf_g that reflects an empirically determined cost increase ci_a (e.g., 15% for PBFT, 0% for Spinning) for each executor that runs on a server without an agreement replica of group g.

Assessing the Performance Capabilities of Servers Prior to being able to assign replica roles, we first need to identify the differences in performance between the servers involved. To achieve this, on each server, we execute a small benchmark that measures the number of MACs the server can calculate per second. This empirical approach has two key advantages: First, it assesses the individual performance capability of a server based on the operation that is the dominant factor with regard to OMADA's overall computing-resource usage. Second, the approach also provides reliable results in cases where the actual performance of a server is not known a priori.

As our assignment algorithm operates with relative performance values, we translate the measured performance numbers into *performance points* reflecting the differences between servers; to have a point of reference, we start by attributing 10 points to the slowest server. To illustrate this step, Fig. 4 shows an example for a heterogeneous setting with five servers in which the two fast servers are able to perform 50 % more MAC calculations per second than the three slow servers. As a consequence, in such a scenario we assign 15 and 10 points to the fast and slow servers, respectively.

Relative Costs for Agreement and Execution To estimate the relative amount of resources OMADA needs to reserve for agreement and execution, we compare the number of MACs each stage computes per client request during normal-case operation. Using PBFT [8] or Spinning [23] as agreement protocol, for example, to order requests in batches of size b an agreement replica must perform $1 + \frac{12f+1}{b}$ MAC calculations per request: 1 for verifying the authenticity of the request, $\frac{10f}{b}$ for ordering it, and $\frac{2f+1}{b}$ for sending EXECUTEs to the executors. In contrast, an executor only calculates $\frac{f+1}{b} + 1$ MACs per request: $\frac{f+1}{b}$ for verifying the EXECUTEs and 1 for authenticating the reply to the client. For $f = 1$ and a batching factor of $b = 10$, this for example means that participating in all agreement groups requires about twice as many computing resources as assuming the role of an executor (e.g., 10 versus 5 performance points for Server 1 in the example in Fig. 4). A similar ratio applies in the optimized case where an agreement group does not need to send EXECUTE messages due to one of its members being collocated with an executor.

Assignment of Roles to Replicas Having determined the individual capabilities of servers as well as the relative costs for agreement and execution, we can derive the mapping of roles to replicas. As greedy mapping algorithms are unable to guarantee optimal solutions and knapsack algorithms in this case involve increased complexity (i.e., the placement of groups with heterogeneous performance characteristics across different servers constitutes a large multi-dimensional knapsack problem), we formulate the problem of mapping roles to replicas as an integer linear program [21], as shown in Fig. 4. In a nutshell, this approach allows us to automatically examine all possible distributions of agreement groups across the servers available in order to find a configuration that maximizes performance. By specifying a number of constraints, we ensure that the selected configuration provides certain properties: First, the configuration allocates an identical amount of resources to all members of the same group to ensure that performance remains stable across group-internal reconfigurations such as view changes (see Fig. 4, the sum in Lines 7–8). Second, it respects the individual performance limits of each server (Lines 7–8). Third, it places executors on the $2f + 1$ most powerful servers, thereby increasing the number of agreement groups that are able to benefit from collocation with an executor (Lines 8–9). Fourth, it does not make use of more than a predefined number of agreement groups to keep the coordination overhead low (Lines 10–11).

Obeying these constraints, the integer linear program assigns individual performance points to each group in the set of possible group-to-server mappings. Based on this result, we can compile the final OMADA configuration by including all groups that received at least one performance point. Furthermore, with each group id representing a particular group-to-server mapping, the result also directly contains the placement of groups. In case the integer linear program produces multiple solutions, we select the solution with the lowest number of groups and the smallest relative performance-point differences between groups to minimize coordination overhead.

Selection of Maximum Batch Sizes To implement performance differences between agreement groups, we define the maximum batch size for each group individually. For a group g, we calculate the maximum batch size by multiplying its performance points with a cost factor cf_g (see Fig. 4) and normalize the result such that the weakest group uses a predefined maximum batch size (e.g., 10). For the configuration in Fig. 4, this for example leads to normalized and rounded maximum batch sizes of $5 * 1 \rightarrow 12$ and $5 * \frac{1}{1+1*0.15} \rightarrow 10$ for Group 1 and Group 2, respectively, which reflects the fact that the executor on Server 3 does not have a collocated replica of Group 2. Using this approach to select maximum batch sizes, less powerful agreement groups process fewer requests to be able to keep up with the more powerful groups. This is necessary as all groups have to handle the same amount of sequence numbers.

4 Implementations

The OMADA system architecture requires agreement groups to establish a stable order on requests but does not make assumptions on how exactly agreement is reached within a group. As a key benefit, this approach offers the flexibility of allowing different BFT protocols to be integrated with OMADA. Below, we present details on two Java-based implementations relying on the PBFT and Spinning protocol, respectively.

OMADA*PBFT* Our first OMADA implementation is based on the PBFT protocol [8] and consequently uses agreement groups consisting of $3f+1$ replicas. Internally, PBFT proceeds in a sequence of views, thereby, based on the view number, for each view deterministically selecting one of the replicas to serve as leader for the group.

As the leader is responsible for starting the agreement process for new requests, information on the identity of the current leader replica must be kept up to date at the client. For this purpose, OMADA$_{\text{PBFT}}$ applies the following approach: Whenever an agreement replica commits a request, the replica includes the number of the current view in the *cri* field of the corresponding EXECUTE to the executors. As discussed in Sect. 3.1, having processed the request an executor then combines the *cri* information from different agreement replicas into a vector and provides this vector to the client as part of the reply. To tolerate faulty agreement replicas, when the client receives a valid reply it determines the view number by choosing the $f+1$ highest value from the vector. If this view number is higher than the view numbers the client has learned from previous replies, the client selects the leader of this view as new contact replica.

With PBFT establishing a stable total order on client requests, the protocol seamlessly integrates with the OMADA system architecture. A key benefit in this context is the fact that executors in OMADA are completely agnostic of the specific protocol used to reach agreement on the sequence numbers of requests. In particular, an executor does not need to be able to interpret the *cri* information contained in EXECUTEs as the executor only concatenates the values it receives. As a consequence, the executor code does not have to be modified for the integration of PBFT (or any other agreement protocol that fulfills the requirements specified in Sect. 3.1) with OMADA.

OMADA*Spinning* In contrast to PBFT, the Spinning protocol [23] does not rely on a fixed leader replica but instead continuously rotates the leader role among the non-faulty replicas in the system to balance load. With the leader role frequently being re-assigned, there is no need in Spinning to maintain knowledge about the identity of the current leader at the client, because each replica is allowed to initiate the agreement protocol for new requests and can therefore serve as contact replica for clients. For this reason, EXECUTEs and REPLYs in OMADA$_{\text{Spinning}}$ do not contain any contact-replica information, causing messages to be slightly smaller than in OMADA$_{\text{PBFT}}$.

Another major difference between PBFT and Spinning concerns ordering parallelism: While PBFT runs multiple protocol instances concurrently, Spinning executes instances in lock step, only starting a new instance after the previous one has finished. Compared with PBFT, Spinning therefore requires larger batching factors to achieve a similar throughput, which is why for OMADA$_{\text{Spinning}}$ we use 50 as maximum batch size for the weakest group (see Sect. 3.2), compared with 10 in OMADA$_{\text{PBFT}}$.

5 Evaluation

In this section, we evaluate OMADA based on a coordination service that relies on our architecture for fault tolerance and comprises a similar interface as ZooKeeper [14]. Coordination services are key building blocks of today's data-center infrastructures as they allow processes of distributed applications to cooperate, for example, by reliably exchanging small chunks of data. As a consequence of being essential for the well-functioning of other applications, it is crucial for coordination services to provide resilience against a wide spectrum of fault scenarios, including Byzantine failures.

5.1 Environment

To compare OMADA with existing approaches, we enable our implementations to also apply the traditional BFT system architecture, executing either plain PBFT or plain Spinning. Using our prototypes, we repeat all experiments with the following settings, all of which are configured to be able to tolerate one Byzantine fault:

– *PBFT-4* and *Spinning-4* make use of the minimum number of replicas required for Byzantine fault tolerance (i.e., four replicas) and thereby represent the typical setting found in most state-of-the-art BFT systems. To order client requests, these implementations execute the PBFT and Spinning protocol, respectively.
– *PBFT-5* and *PBFT-6* are variants of PBFT-4 with five and six replicas, respectively, each running on a separate server; accordingly, *Spinning-5* and *Spinning-6* represent variants of Spinning-4. We evaluate these settings as they allow us to study the effects of introducing additional resources into traditional BFT systems.
– OMADA$_{\text{PBFT}}$ and OMADA$_{\text{Spinning}}$, as described in Sect. 4, rely on our novel BFT system architecture and are distributed across up to six servers.

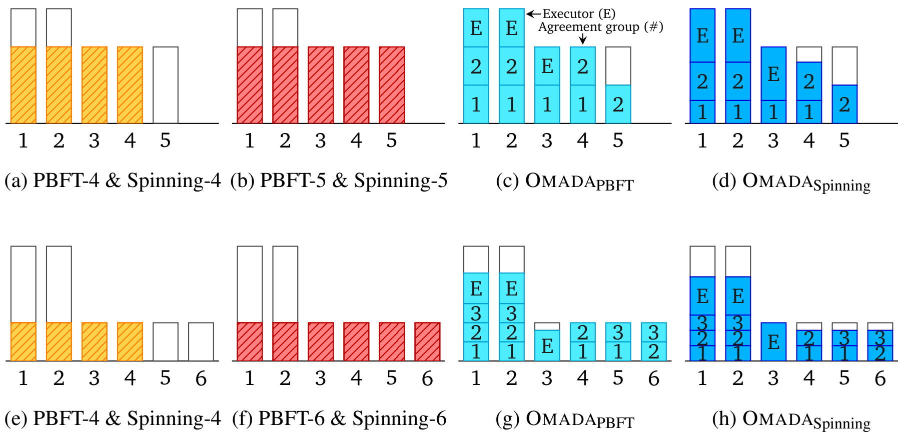


Fig. 5 System configurations for the two heterogeneous settings used in the experimental evaluation, comprising five servers (top) and six servers (bottom) with different performance capabilities, respectively.

In order to be able to investigate the influence of heterogeneity, we conduct our experiments using two different settings of servers with non-uniform performance capabilities, as illustrated in Fig. 5. In this context, we distinguish between two categories of machines: *fast* servers and *slow* servers; the difference in performance achieved varies between experiments and will therefore be explained in later sections. All servers use Ubuntu 16.04 LTS as operating system along with OpenJDK 8u131 and are connected via switched Gigabit Ethernet.

For each experiment, we vary the number of clients writing data to the coordination service in chunks of typical sizes of 128 bytes. Besides, we have also conducted experiments evaluating read operations, but we omit these results due to limited space and because they offer similar insights as the write results. To generate the workloads, we execute the clients on a separate server and distribute them across the available agreement groups in a way so that more powerful groups handle more clients. During an experiment, each client runs in a closed loop, that is, it only sends a new request after having obtained a stable reply for its previous one. Each data point presented in the following represents the average over 5 runs.

5.2 Exploiting Additional Computing Resources

The first heterogeneous setting we use for our experiments comprises two fast and three slow servers (see Fig. 5a–5d). The fast servers are equipped with Intel Xeon E5645 CPUs (2.4 GHz) and 32 GB RAM, whereas the slow servers have Intel Xeon E5520 CPUs (1.6 GHz) and 8 GB RAM. Based on the rate of MAC calculations per second, a slow server in this setting achieves about two thirds of the performance of a fast server. For this environment, OMADA$_{\text{PBFT}}$'s group assignment procedure creates the configuration we already used as example to explain the procedure in Sect. 3.2. The configuration comprises two agreement groups with maximum batch sizes of 12 and 10, respectively. For a fair comparison, we configure PBFT-4 and PBFT-5 to use a maximum batch size of 11, which is the average batch size of the two OMADA$_{\text{PBFT}}$ groups. As discussed in Sect. 4, the maximum batch sizes for the two agreement groups in OMADA$_{\text{Spinning}}$ are larger (i.e., 50 and 83, respectively) to compensate for the fact that Spinning executes protocol instances in lock step. Accordingly, we configure a maximum batch size of 67 for Spinning-4 and Spinning-5.

Fig. 6 presents the measured latency and throughput for this experiment. All six evaluated systems achieve low latency until reaching saturation, at which point latency rises quickly when the workload is increased further. In general, the latency of PBFT-5 is higher than the latency provided by PBFT-4; the same holds for Spinning-5 compared with Spinning-4. This is caused by the larger quorum sizes and thus the additional messages that are necessary to include the fifth server, as well as the larger MAC authenticators which grow in size and require an additional MAC calculation. Compared with PBFT-4, the latency of OMADA$_{\text{PBFT}}$ is slightly higher which can be attributed to the increased coordination overhead necessary to manage the two agreement groups. However, unlike PBFT-4, its group-based system architecture enables OMADA$_{\text{PBFT}}$ to sustain low latency at higher throughputs by effectively utilizing the fifth server. OMADA$_{\text{Spinning}}$ shows a similar picture compared with Spinning-4.

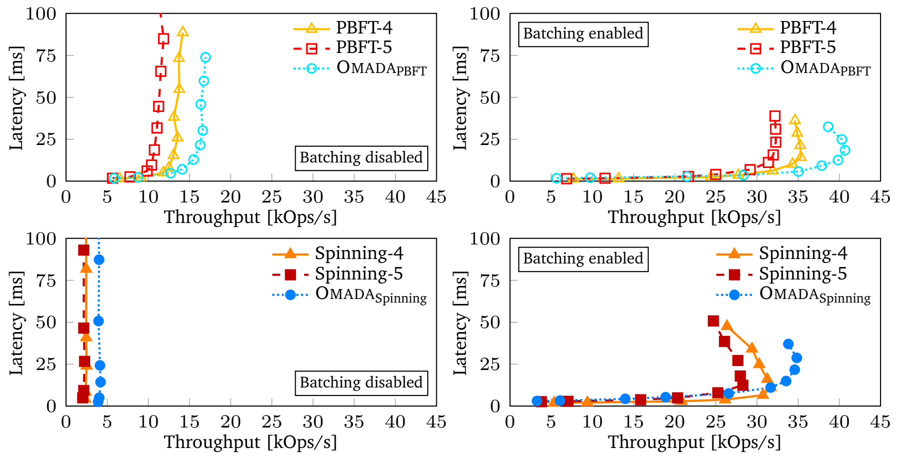


Fig. 6 Relationship between throughput and latency for the heterogeneous setting with two fast and three slow servers using PBFT (top) and Spinning (bottom) as agreement protocol, respectively.

With regard to the maximum throughput achievable, our experiments confirm the key advantage of the batching optimization, which enables all six evaluated systems to provide a significantly higher performance, independent of the specifics of their agreement protocols. Nevertheless, due to not being able to benefit from the additional server, the maximum throughput of PBFT-5 is still about 9% lower than the maximum throughput of PBFT-4; without batching the difference is about 16%. In contrast, OMADA$_{PBFT}$ exploits the additional computing resources offered by the fifth server and therefore compared with PBFT-4 achieves an increase in maximum throughput of 15% when batching is enabled and 19% when batching is disabled.

For Spinning-4 without batching, throughput performance is limited by the protocol executing agreement instances sequentially. Providing a throughput of only about 2.5 kOps/s, a large amount of resources remains unused in this setting. Our measurement results show that Spinning-5's traditional architecture prevents the system from taking advantage of this fact. Instead, the additional transmission and authentication overhead in Spinning-5 leads to a throughput decrease of 10% compared with Spinning-4. Relying on two agreement groups and running them in parallel, OMADA$_{Spinning}$ on the other hand is able to use parts of the spare resources and therefore achieves a throughput increase of nearly 68% compared with Spinning-5.

5.3 Assessing the Costs of Groups

To evaluate the group-coordination overhead in OMADA in more detail, we use the same setup as in the previous experiment but now vary the number of agreement groups by splitting each of the two existing groups into up to four smaller ones. For each of the two OMADA systems, this yields four configurations comprising between two and eight agreement groups. The measurement results presented in Fig. 7 show that the overhead for operating additional agreement groups in OMADA is small but

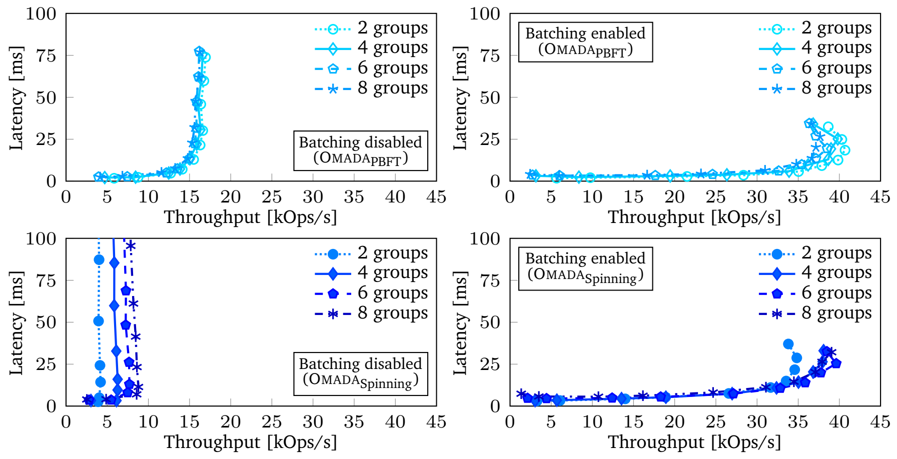


Fig. 7 Relationship between throughput and latency in OMADA$_{\text{PBFT}}$ (top) and OMADA$_{\text{Spinning}}$ (bottom) with different numbers of groups for the heterogeneous setting with two fast and three slow servers.

measurable. With agreement groups in OMADA being largely independent of each other, having fewer groups has the advantage that it becomes less likely that requests of one group need to wait at an executor until requests with lower sequence numbers of another group become ready for execution. As a consequence, with two agreement groups OMADA provides lower latency than with eight agreement groups. For OMADA$_{\text{PBFT}}$, a similar effect can also be observed with regard to throughput: The maximum throughput of two-group OMADA$_{\text{PBFT}}$ without batching for example is about 4% higher than the maximum throughput of eight-group OMADA$_{\text{PBFT}}$. For OMADA$_{\text{Spinning}}$, the costs associated with additional agreement groups in some cases are outweighed by the positive effects of the increased parallelism at the agreement stage. In particular, this is true when batching is disabled. As our results show, in such settings throughput increases with the number of agreement groups, which for example results in OMADA$_{\text{Spinning}}$ being able to process almost 110% more requests per second with eight agreement groups than with two agreement groups.

5.4 Evaluating the Impact of Faults

In our third experiment, we evaluate the impact of a server failure on the performance of OMADA. As illustrated in Figure 8, when one of the fast servers hosting a leader replica crashes 60 seconds into the experiment, the affected agreement group in OMADA needs to elect a new leader before being able to make progress again. Relying on the built-in view-change mechanism of the underlying BFT agreement protocol for this purpose, this process takes about the same time in OMADA$_{\text{PBFT}}$ and OMADA$_{\text{Spinning}}$ as it does in PBFT and Spinning, respectively. With one fast server and three slow servers remaining in the system after the crash, OMADA's throughput still matches (in case of PBFT) or even exceeds (in case of Spinning) the performance provided by the two baseline systems under fault-free conditions.

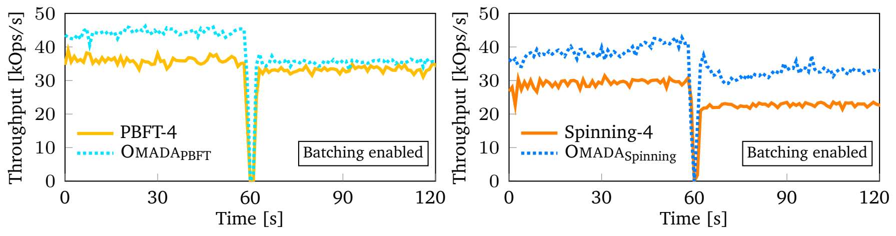


Fig. 8 Impact of a server failure on performance in OMADA$_{PBFT}$ (left) and OMADA$_{Spinning}$ (right).

5.5 Analyzing the Effects of Heterogeneity

For our final experiment, we use a heterogeneous environment that differs from the previous setting and enables us to study the adaptability of OMADA. As shown in Fig. 5e–5h, the systems now comprise an additional server. Furthermore, by limiting the number of MACs a slow server is able to calculate per second, we create a scenario in which a slow server only achieves a third of the performance of a fast server. In practice, such performance differences between replicas can be the result of not only incorporating servers with different capabilities but also relying on heterogeneous replica implementations to minimize the probability of common mode failures. For this purpose, replicas for example may make use of different programming languages or operating systems to reduce the fault dependency between them [13]. In this experiment, OMADA$_{PBFT}$ and OMADA$_{Spinning}$ both have three agreement groups that are distributed across servers as shown in Fig. 5g and Fig. 5h, respectively. As all groups in OMADA$_{PBFT}$ use a maximum batch size of 10 we select the same size for PBFT-4 and PBFT-6; all Spinning variants order batches of at most 50 requests.

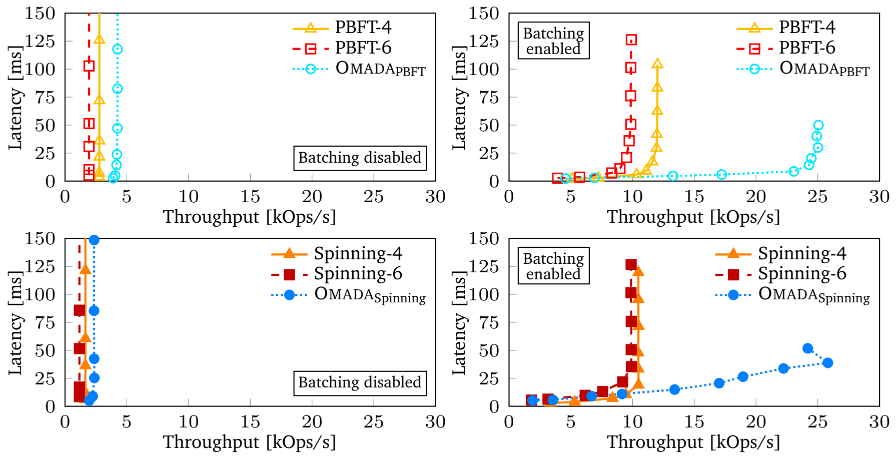


Fig. 9 Relationship between throughput and latency for the heterogeneous setting with two fast and four slow servers using PBFT (top) and Spinning (bottom) as agreement protocol, respectively.

The results in Fig. 9 show that as an effect of the reduced amount of computing resources, the maximum throughputs achieved in this experiment are lower than the maximum throughputs in previous experiments. In particular, both PBFT-4 and PBFT-6 are unable to utilize most of the resources available on the fast servers due to being limited by the slow servers. With a decreased maximum throughput of 30% (without batching) and 18% (with batching) compared with PBFT-4, PBFT-6 performs significantly worse than its counterpart PBFT-5 in Sect. 5.2, which confirms the system's lack of scalability. Similarly, there is a notable throughput decrease for Spinning-6 in comparison to Spinning-4 (i.e., 29% without batching and 6% with batching) and a significant difference to Spinning-5 in our first experimental setting. OMADA, on the other hand, not only benefits from the additional servers but also utilizes a large part of the computing resources on the fast servers by enabling their replicas to act as executors and to furthermore participate in all three agreement groups. This way, when batching is disabled $\text{OMADA}_{\text{PBFT}}$ achieves a maximum throughput that is 53% higher than the maximum throughput of PBFT-4 for this experiment; similarly, $\text{OMADA}_{\text{Spinning}}$ shows an improvement of 44% over Spinning-4. As in previous experiments, enabling the batching optimization has a positive impact on overall maximum throughput also for this evaluation setting. With batching enabled, $\text{OMADA}_{\text{PBFT}}$ provides a 108% higher throughput performance than PBFT-4. $\text{OMADA}_{\text{Spinning}}$ even shows an improvement of 146% over Spinning-4, thereby confirming the effectiveness of our approach in heterogeneous environments.

6 Related Work

Yin et al. [25] proposed a BFT system architecture (in the following referred to as SAfE) that separates agreement from execution and comprises a dedicated cluster of replicas for each of the two stages. OMADA builds on this idea by splitting the responsibilities for ordering and executing requests into different roles and allowing each replica to assume one or more of these roles depending on its performance capabilities. As a consequence, both SAfE and OMADA need to provide the agreement stage with means to prove to the execution stage that a request has been committed. In SAfE, the agreement cluster for this purpose sends internal protocol messages to the execution cluster, thereby (1) creating a tight coupling between both stages and (2) requiring agreement messages to be authenticated with additional MACs in order to be verifiable by the execution cluster. In contrast, OMADA cleanly decouples the agreement protocol from the execution protocol (i.e., the transmission of EXECUTEs) to keep the authentication cost for agreement messages low. In addition, OMADA saves further network and computing resources by suppressing an agreement group's EXECUTEs if an executor is collocated with an agreement replica of the group.

UpRight [10] goes one step further than SAfE and, besides agreement and execution, relies on a third stage responsible for receiving and buffering requests. Using this stage, the system can forward large requests directly to the execution while performing the agreement on their hashes, thereby reducing the load on the agreement cluster at the cost of increased latency. Similar to SAfE and PBFT, UpRight's agreement stage comprises $3f+1$ replicas and is unable to benefit from additional servers.

Amir et al. [2] proposed a hierarchical replication architecture for wide-area environments in which each geographical site hosts a group of replicas executing a local agreement protocol. At the global level, each of these groups acts a participant in a wide-area replication protocol. Compared to this work, OMADA also merges the output of multiple agreement groups into a global total order, however, executors in OMADA for this purpose do not require a full-fledged replication protocol.

Kapritsos and Junqueira [17] outlined a general approach to partition the agreement workload in order to improve the scalability of replicated systems. In particular, they presented a crash-tolerant protocol that, similar to OMADA, is able to assign different parts of the sequence-number space to different replica groups. Having been designed to provide resilience against crashes, unlike OMADA, the protocol however cannot ensure liveness in the presence of arbitrary replica failures. Furthermore, their approach does not address replicated systems that consist of heterogeneous servers.

The idea of using multiple replicas to independently order requests that are then merged into a single ordered request stream has been explored in various ways in the context of crash fault tolerance. In the accelerated ring protocol developed by Babay and Amir [5] the replicas pass on a single token after proposing several requests while compensating for network latency. Aguilera and Strom [1] presented an algorithm to deterministically merge multiple message streams primarily based on the timestamps of individual messages. Mencius [19], a crash-tolerant protocol for wide-area networks, evenly partitions the sequence numbers across all replicas.

COP [6] and Sarek [18] parallelize the handling of agreement-protocol instances within each replica of a BFT system to effectively utilize multi-core servers. Focusing on the internal structure of a replica, these approaches are orthogonal to the replication scheme presented in this paper and could therefore also be applied to OMADA.

The few works that have studied heterogeneity in BFT systems [9, 11] use heterogeneous execution-stage implementations to reduce the probability that a single fault causes multiple replica failures. OMADA, in contrast, deals with the consequences of heterogeneity at the level of the entire system. This enables OMADA to exploit performance capabilities that so far have not been used, following the principle that it is better to harness the differences between replicas than to try to compensate them.

7 Conclusion

OMADA is a BFT system architecture that is able to use additional servers by partitioning the agreement stage into multiple largely independent groups. In environments comprising servers with heterogeneous performance capabilities, OMADA tailors the distribution of the agreement groups to the set of servers available in order to exploit the individual performance capabilities of each server. Our evaluation has shown that in contrast to existing systems OMADA is able to benefit from additional computing resources, and that our approach is particularly effective in heterogeneous settings with a significant performance difference between fast and slow servers.

Acknowledgements This work was partially supported by the German Research Council (DFG) under grant no. DI 2097/1-2 (“REFIT – Resource-Efficient Fault and Intrusion Tolerance”).

References

1. Aguilera MK, Strom RE (2000) Efficient atomic broadcast using deterministic merge. In: Proc. of the 19th Symp. on Principles of Distributed Computing, pp 209–218
2. Amir Y, Coan B, Kirsch J, Lane J (2007) Customizable fault tolerance for wide-area replication. In: Proc. of the 26th Int'l Symp. on Reliable Distributed Systems, pp 65–82
3. Amir Y, Coan B, Kirsch J, Lane J (2011) Prime: Byzantine replication under attack. IEEE Transactions on Dependable and Secure Computing 8(4):564–577
4. Aublin PL, Mokhtar SB, Quéma V (2013) RBFT: Redundant Byzantine fault tolerance. In: Proc. of the 33rd Int'l Conference on Distributed Computing Systems, pp 297–306
5. Babay A, Amir Y (2016) Fast total ordering for modern data centers. In: Proc. of the 36th Int'l Conference on Distributed Computing Systems, pp 669–679
6. Behl J, Distler T, Kapitza R (2015) Consensus-oriented parallelization: How to earn your first million. In: Proc. of the 16th Middleware Conference, pp 173–184
7. Bessani A, Sousa J, Alchieri EEP (2014) State machine replication for the masses with BFT-SMaRt. In: Proc. of the 44th Int'l Conference on Dependable Systems Networks, pp 355–362
8. Castro M, Liskov B (1999) Practical Byzantine fault tolerance. In: Proc. of the 3rd Symp. on Operating Systems Design and Implementation, pp 173–186
9. Castro M, Rodrigues R, Liskov B (2003) BASE: Using abstraction to improve fault tolerance. ACM Transactions on Computer Systems 21(3):236–269
10. Clement A, Kapritsos M, Lee S, Wang Y, Alvisi L, Dahlin M, Riche T (2009) UpRight cluster services. In: Proc. of the 22nd Symp. on Operating Systems Principles, pp 277–290
11. Distler T, Kapitza R, Reiser HP (2010) State transfer for hypervisor-based proactive recovery of heterogeneous replicated services. In: Proc. of the 5th "Sicherheit, Schutz und Zuverlässigkeit" Conference, pp 61–72
12. Distler T, Cachin C, Kapitza R (2016) Resource-efficient Byzantine fault tolerance. IEEE Transactions on Computers 65(9):2807–2819
13. Garcia M, Bessani A, Gashi I, Neves N, Obelheiro R (2014) Analysis of operating system diversity for intrusion tolerance. Software—Practice & Experience 44(6):735–770
14. Hunt P, Konar M, Junqueira F, Reed B (2010) ZooKeeper: Wait-free coordination for Internet-scale systems. In: Proc. of the 2010 USENIX Annual Technical Conference, pp 145–158
15. Junqueira F, Bhagwan R, Hevia A, Marzullo K, Voelker GM (2005) Surviving Internet catastrophes. In: Proc. of the 2005 USENIX Annual Technical Conference, pp 45–60
16. Kapitza R, Behl J, Cachin C, Distler T, Kuhnle S, Mohammadi SV, Schröder-Preikschat W, Stengel K (2012) CheapBFT: Resource-efficient Byzantine fault tolerance. In: Proc. of the 7th European Conference on Computer Systems, pp 295–308
17. Kapritsos M, Junqueira FP (2010) Scalable agreement: Toward ordering as a service. In: Proc. of the 6th Workshop on Hot Topics in System Dependability, pp 7–12
18. Li B, Xu W, Abid MZ, Distler T, Kapitza R (2016) SAREK: Optimistic parallel ordering in Byzantine fault tolerance. In: Proc. of the 12th European Dependable Computing Conference, pp 77–88
19. Mao Y, Junqueira FP, Marzullo K (2008) Mencius: Building efficient replicated state machines for WANs. In: Proc. of the 8th Conference on Operating Systems Design and Implementation, pp 369–384
20. Ou Z, Zhuang H, Lukyanenko A, Nurminen JK, Hui P, Mazalov V, Ylä-Jääski A (2013) Is the same instance type created equal? Exploiting heterogeneity of public clouds. IEEE Transactions on Cloud Computing 1(2):201–214
21. Papadimitriou CH, Steiglitz K (1998) Combinatorial Optimization: Algorithms and Complexity. Dover Publications
22. Pease M, Shostak R, Lamport L (1980) Reaching agreement in the presence of faults. Journal of the ACM 27(2):228–234
23. Veronese GS, Correia M, Bessani AN, Lung LC (2009) Spin one's wheels? Byzantine fault tolerance with a spinning primary. In: Proc. of the 28th Int'l Symp. on Reliable Distributed Systems, pp 135–144
24. Veronese GS, Correia M, Bessani AN, Lung LC (2010) EBAWA: Efficient Byzantine agreement for wide-area networks. In: Proc. of the 12th Symp. on High-Assurance Systems Engineering, pp 10–19
25. Yin J, Martin JP, Venkataramani A, Alvisi L, Dahlin M (2003) Separating agreement from execution for Byzantine fault tolerant services. In: Proc. of the 19th Symp. on Operating Systems Principles, pp 253–267