6-2009

Design and Implementation of a Byzantine Fault Tolerance Framework for Web Services

Wenbing Zhao
Cleveland State University, w.zhao1@csuohio.edu

Publisher's Statement

NOTICE: this is the author's version of a work that was accepted for publication in Journal of Systems and Software. Changes resulting from the publishing process, such as peer review, editing, corrections, structural formatting, and other quality control mechanisms may not be reflected in this document. Changes may have been made to this work since it was submitted for publication. A definitive version was subsequently published in Journal of Systems and Software, 82, 6, (06-01-2009); 10.1016/j.jss.2008.12.037

Original Citation

Zhao, W. (2009). Design and implementation of a Byzantine fault tolerance framework for Web services. The Journal of Systems & Software, 82(6), 1004-1015. doi:10.1016/j.jss.2008.12.037

Repository Citation

Design and implementation of a Byzantine fault tolerance framework for Web services ☆

Wenbing Zhao *

Department of Electrical and Computer Engineering, Cleveland State University, 2121 Euclid Ave, Cleveland, OH 44115, United States

1. Introduction

Driven by business needs and the availability of the latest Web services technology, we have seen increasing reliance on services provided over the Web. This undoubtedly has increased the dependability requirement on these services. Recognizing this need, the Web services community has proposed the Web Services Reliable Messaging (WS-RM) specification (Bilorusets et al., 2005) and it is recently ratified by OASIS (Organization for the Advancement of Structured Information Standards). WS-RM is a very good starting point to increase the reliability of Web services interactions and furthermore it has been widely supported by many commercial and open-source frameworks.[1]

However, for many mission-critical Web services, WS-RM might be inadequate to meet the high reliability needs. First of all, WS-RM does not guarantee the high availability of Web services, which would require the use of space redundancy, i.e., the Web services must be replicated. Second, considering the untrusted communication environment in which these services operate, there are legitimate concerns on the security of the Web services because if a Web service is compromised by an adversary, not only may it be made unavailable, perhaps more seriously, it may be rendered to provide false/invalid information to the clients. Furthermore, when the service is replicated, a new type of attacks could arise, i.e., both a faulty client and a faulty server replica could disseminate conflicting information to different replicas, aiming to destroy strong replica consistency, which is an essential requirement for state machine replication (Schneider, 1990).

To control these types of threats, the arbitrary fault model must be adopted. An arbitrary fault, often referred to as a Byzantine fault (Lamport et al., 1982), encompasses both a benign fault such as a crash fault, and a malicious fault imposed by an adversary. In the presence of Byzantine faults, WS-RM cannot guarantee the integrity of the Web services, e.g., a compromised Web service may not adhere to the exactly-once delivery policy even if the Web service is configured to do so. As recognized by many researchers (Castro and Liskov, 1999; Yin et al., 2003), Byzantine fault tolerance (BFT) seems to be a promising approach to achieving highly secure and reliable Web services.

In this article, we describe such a Byzantine fault tolerance framework for Web services, referred to as BFT-WS in this paper. Our framework differs from similar work primarily in the following two aspects:

(1) BFT-WS is backward compatible with WS-RM. Due to the widespread adoption of WS-RM, we anticipate that many reliability-aware Web services are already supporting WS-RM. It seems to be natural to upgrade these Web services

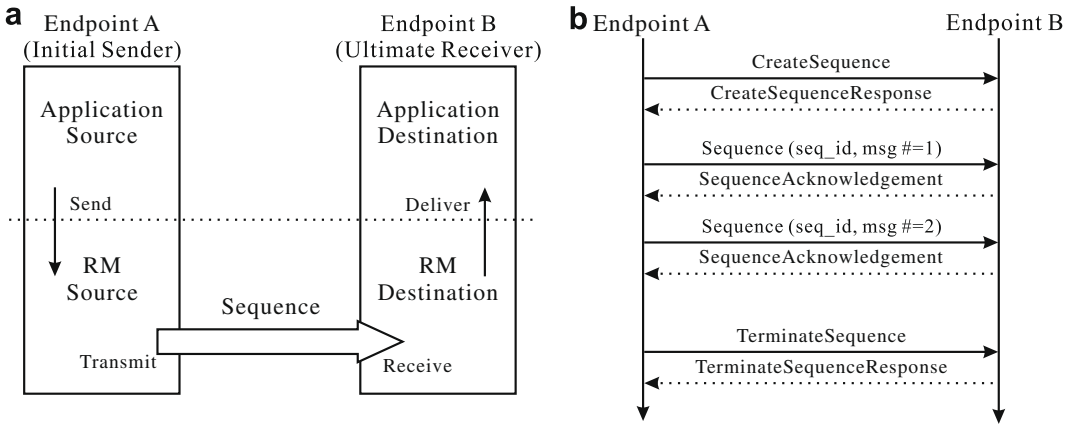

Fig. 1. (a) The reliable messaging model in WS-RM. (b) The reliable messaging protocol in WS-RM.

for Byzantine fault tolerance. The compatibility of our framework with WS-RM means that these Web services would need virtually no changes except the addition of state retrieval and restoration handlers. We believe that this would ease the adoption of Byzantine fault tolerance technology for Web services.

(2) Our framework is based entirely on Web services core technology (such as SOAP and WSDL) and the WS-* standards, without resorting to any proprietary communication protocols or APIs.

BFT-WS is implemented on top of Sandesha2,[2] which is an open-source implementation of the WS-RM standard for Apache Axis2 in Java. In BFT-WS, all fault tolerance mechanisms operate on top of the standard SOAP messaging framework for maximum interoperability. BFT-WS inherits Sandesha2's pluggability, and hence, it requires minimum changes to the Web applications (both the client and the service sides). The core fault tolerance mechanisms in BFT-WS are based on the well-known Castro and Liskov's BFT algorithm (Castro and Liskov, 1999). The framework also incorporated the idea of the separation of agreement (on the total ordering of messages) and execution introduced by Yin et al. (2003) so that the BFT-WS framework can be used in several different configurations, and thereby increasing the flexibility of BFT-WS for use in practical systems.

The performance evaluation of BFT-WS has been carefully conducted. The results show that the BFT-WS framework indeed introduces only moderate runtime overhead verses the original Sandesha2 framework considering the complexity of the Byzantine fault tolerance mechanisms.

This paper is structured as follows. Section 2 introduces some necessary background information. Section 3 describes the design rationale, the system models, and the architecture of BFT-WS. Section 4 presents the performance evaluation results. Section 5 describes related work, and Section 6 concludes the paper.

2. From reliable messaging to Byzantine fault tolerance

2.1. Web services reliable messaging

The Web services reliable messaging (WS-RM) standard describes a reliable messaging (RM) protocol between two endpoints, termed as RM source (RMS) and RM destination (RMD). The reliable messaging model in the WS-RM standard and an example of the reliable messaging protocol are shown in Fig. 1a and b, respectively. The core concept introduced in WS-RM is *sequence*. A sequence is a unidirectional reliable channel between the RMS and the RMD. As shown in Fig. 1b, at the beginning of a reliable conversation between the two endpoints, a unique sequence (identified by a unique sequence ID) must first be created (through the CreateSequence request and response). The sequence is terminated when the conversation is over (through the TerminateSequence request and response). For each message sent over the sequence, a unique message number must be assigned to it. The message number starts at 1 and is incremented by 1 for each subsequent message. The reliability of the messaging is achieved by the retransmission and positive acknowledgement mechanisms. At the RMS, a message sent is buffered and retransmitted until the corresponding acknowledgement from the RMD is received, or until a predefined retransmission limit has been exceeded. For efficiency reason, the RMD might not send acknowledgement immediately upon receiving an application message, and the acknowledgements for multiple messages can be piggybacked with another application message in the response sequence, or be aggregated in a single explicit acknowledgement message.

Because it is quite common for two endpoints to engage in two-way communications, the RMS can include an Offer element in its CreateSequence request to avoid an explicit new sequence establishment step for the traffic in the reverse direction.

Furthermore, WS-RM defines a set of delivery assurances, including AtMostOnce, AtLeastOnce, ExactlyOnce, and InOrder. The meaning of these assurances are self-explanatory. The InOrder assurance can be used together with any of the first three assurances. The strongest assurance is ExactlyOnce combined with InOrder delivery.

The WS-RM standard has been widely supported and there exist many implementations, most of which are commercial. We choose to use Sandesha2 for this research, due to its open-source nature and its support for Axis2, the second generation open-source SOAP engine that enables pluggable modules.

2.2. Byzantine fault tolerance

Byzantine fault tolerance (BFT) refers to the capability of a system to tolerate Byzantine faults. In a client–server system, it can be achieved by replicating the server and by ensuring all server replicas to execute the same request in the same order. The latter means that the server replicas must reach an agreement on the set of requests and their relative ordering despite Byzantine faulty replicas and clients. Such an agreement is often referred to as Byzantine agreement (Lamport et al., 1982).

Byzantine agreement algorithms had been too expensive to be practical until Castro and Liskov invented the BFT algorithm men-

[2] The Apache Sandesha2 project, http://ws.apache.org/sandesha/sandesha2/.

tioned earlier (Castro and Liskov, 1999). The BFT algorithm is designed to support client–server applications running in an asynchronous distributed environment with the Byzantine fault model. The implementation of the algorithm contains two parts. At the client-side, a lightweight library is responsible to send the client's request to the primary replica, to retransmit the request to all server replicas on the expiration of a retransmission timer (to deal with the primary faults), and to collect and vote on the replies. The main BFT algorithm is executed at the server-side by a set of $3f+1$ replicas to tolerate f Byzantine faulty replicas. One of the replicas is designated as the primary while the others are backups.

In the BFT framework, a replica is modeled as a state machine. The replica is required to run (or rendered to run) deterministically. The state change is triggered by remote invocations on the methods offered by the replica.

As shown in Fig. 2, the normal operation of the (server-side) BFT algorithm involves three phases. During the first phase (called pre-prepare phase), the primary multicasts a pre-prepare message containing the client's request, the current view and a sequence number assigned to the request to all backups.

A backup verifies the request message and the ordering information. If the backup accepts the message, it multicasts to all other replicas a prepare message containing the ordering information and the digest of the request being ordered. This starts the second phase, i.e., the prepare phase. A replica waits until it has collected $2f$ prepare messages from different replicas (including the message it has sent if it is a backup) that match the pre-prepare message before it multicasts a commit message to other replicas, which starts the third phase (i.e., commit phase). The commit phase ends when a replica has received $2f$ matching commit messages from other replicas. At this point, the request message has been totally ordered and it is ready to be delivered to the server application if all previous requests have already been delivered.

The BFT framework uses a number of optimizations to improve the runtime performance under normal operation. The most relevant optimization related to this work is the batching mechanism. When batching is enabled, the primary postpones total ordering of a request until there are already k batches of messages being ordered, where k is a tunable parameter and it is often set to 1. When the primary is ready to order a new batch of messages, it assigns the next sequence number for the entire group of application requests. This mechanism could significantly improve the system throughput under heavy request load.

For garbage collection, each replica periodically takes a snapshot of its state (referred to as a checkpoint) and multicasts a checkpoint message including the sequence number n of the last request whose execution is reflected in the checkpoint, the digest of the checkpoint d, and the replica id. A checkpoint becomes *stable* when a replica has collected $2f+1$ checkpoint messages for the same sequence number n with the same digest d signed by different replicas. At this point, the replica can discard all control messages (such as pre-prepare, prepare and commit messages) with sequence number less than or equal to n, and all earlier checkpoints and checkpoint messages. A stable checkpoint is also useful to bring a slow replica up-to-date.

If the primary or the client is faulty, a Byzantine agreement on the ordering of a request might not be reached, in which case, a new view is initiated, triggered by a timeout on the current view. A different primary is designated in a round-robin fashion for each new view installed.

Since the publication of the seminal work of Castro and Liskov (1999), a number of alternative BFT algorithms have been proposed (Cowling et al., 2006; Abd-El-Malek et al., 2005; Kotla et al., 2007), each offers better performance in some circumstances. Among them, HQ (Cowling et al., 2006) and QU (Abd-El-Malek et al., 2005) both assume that the BFT infrastructure knows if a request will update the server state. In practice, however, it might not be feasible for a generic BFT infrastructure to gain such knowledge unless it is customized to run a specific application. Despite the fact that Zyvzzya (Kotla et al., 2007) improves the runtime performance significantly during normal operation (i.e., when there is no fault) in most cases, the recovery mechanisms (needed to handle primary failures) are considerably more complicated and we are not aware of any implementation of the complete Zyvzzya algorithm. Therefore, we choose to use the Castro and Liskov's BFT algorithm in our framework. In particular, the availability of a publicly-accessible C++ implementation of Castro and Liskov's algorithm is instrumental for us to build a working prototype of BFT-WS in a reasonable time-frame.

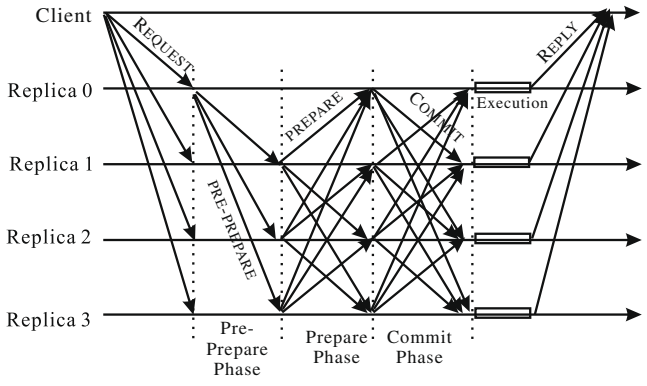

Fig. 2. Normal operation of the BFT algorithm.

3. Design and implementation of BFT-WS

In this section, we first elaborate the rationale on the design of BFT-WS. Then, we present the system models and the architecture of our framework.

3.1. Rationale for our approach

Based on our previous experiences in building fault-tolerant CORBA frameworks (Zhao et al., 2004), we classify typical approaches to building Byzantine fault tolerance (BFT) middleware frameworks into the following three categories.

(1) *Application Programming Interface (API) Based Approach*. In this approach, a Web server application must program the fault tolerance logic, such as replication group creation, join and leave, explicitly into the application logic. The main benefit of this approach is that fault tolerance itself can be provided as a high-level Web service, which is very desirable for service-oriented computing. However, this approach has obvious drawbacks: (1) mixing fault tolerance logic and application logic increases the complexity of the application design and implementation, which might lead to more buggy software, instead of achieving better robustness against faults, and (2) it might be very difficult to retrofit existing applications for fault tolerance using these APIs, especially when the source code is not available. Due to the above concerns, we prefer not to follow this approach.

(2) *Interception Based Approach*. In this approach, incoming and outgoing messages to and from a Web service, respectively, are intercepted at the sender. The intercepted requests are then totally ordered prior to their delivery to the Web ser-

vice application via a group communication system (Salas et al., 2006), or a consensus-based algorithm (Castro and Liskov, 1999). The interception layer also injects additional mechanisms for fault tolerance, such as state synchronization among the replicas and voting at the client. Interception is typically achieved by using a separate proxy process, as in Salas et al. (2006), or by using an under-the-SOAP-engine hook, as in Merideth et al. (2005). The benefit of this approach is that existing and future Web services can be rendered fault-tolerant nearly transparently (with the exception that the application must provide a way for the fault tolerance framework to retrieve and restore the application state, which is a common requirement for all fault tolerance frameworks). However, the main issue of this approach lies in its use of proprietary messaging protocols. This is incompatible with the design principles of Web services, which call for transport independence and mandate SOAP-based communications. The use of proprietary messaging protocols compromises the interoperability of Web services. Secondly, the use of a separate proxy process or an under-the-SOAP-engine hook often leads to repeated parsing of SOAP messages (as we will show later in this paper, not all SOAP messages should be totally ordered, and these messages must be identified prior to the total ordering operation), which can lead to performance degradation.

(3) *Integration Based Approach*. In this approach, the fault tolerance mechanisms are built directly into the SOAP engine and only standard Web services messaging and transport protocols are used. For this approach to be practical, it requires that the SOAP engine support pluggable modules for user customization. Fortunately, the Apache Axis2 SOAP engine,[3] the most popular open-source SOAP engine, satisfies this requirement. We believe that this approach possesses the benefits of the other two approaches while avoiding their drawbacks:

- ←Fault tolerance in effect is still provided as a Web service. The difference between the integration based approach and the API based approach is that, in the integration based approach, the fault tolerance Web service operations are invoked not directly by the Web application, but by the fault tolerance modules plugged into the SOAP engine, which avoids the problem of the API based approach.
- ←The benefit of fault tolerance transparency in the interception based approach is obviously satisfied as well because the invocations on the fault tolerance services are carried out transparently by the plugin modules. The only obligation from the user is to specify the fault tolerance parameters such as the replication degree in a configuration file, which can be changed during the deployment phase. Furthermore, because the fault tolerance mechanisms are built into the plugin modules in the integration based approach, standard Web services messaging and transport protocols are used for replica coordination and recovery, and hence, avoiding the problems of the interception based approach.

We believe that any type of middleware for Web services must use standard Web services technologies and must follow the design principles of Web services, and fault tolerance middleware for Web services is no exception. Therefore, we decide to follow the integration based approach to develop a Byzantine fault tolerance framework for Web services. Furthermore, we strive to build a framework that not only uses basic Web services techniques such as SOAP and WSDL, but is compatible with existing WS-* specifications as well (Web services would not have been widely adopted without these WS-* specifications). In particular, our framework is designed to be backward compatible with the WS-RM standard (Bilorusets et al., 2005) because Byzantine fault tolerance is logically the next step beyond reliable messaging for mission-critical Web services. By offering this backward compatibility, we hope to make it possible to upgrade existing WS-RM compliant applications for Byzantine fault tolerance with minimum modifications.

3.2. System models

We assume that the Web services applications operate in an asynchronous distributed environment. To ensure liveness, it is necessary to assume certain synchrony, i.e., the message transmission and processing delay has an asymptotic upper bound (Castro and Liskov, 1999). This bound is dynamically explored in the BFT algorithm in that each time a view change occurs, the timeout for the new view is doubled. We assume that the synchronous communication style is used by the Web services application, i.e., a client (or a middle-tier server) would issue a request (or a nested request) to a server and then is blocked waiting for the corresponding reply.

Both the client and the server can be Byzantine faulty, i.e., they are subject to arbitrary faults (both hardware and malicious faults). To achieve Byzantine fault tolerance, the server is replicated with $3f + 1$ replicas to tolerate up to f faulty nodes. We assume that the server replicas are sufficiently diversified so that they fail independently under Byzantine attacks. A common approach to achieving diversity is to employ the n-version programming technique (Chen and Avizienis, 1995) where each replica is implemented according to a different design and possibly using a different programming language. For some applications, such as the networked file system, there already exist many off-the-shelf different implementations, which could readily be used for Byzantine fault tolerance (Castro et al., 2003).

Each replica is modeled as a state machine, consequently, we assume that the application would run (or would be rendered to run) deterministically. The state change is triggered by remote invocations on the methods offered by the replica. We are fully aware that practical applications often exhibit non-deterministic behaviors. How to render such applications to run deterministically has been studied by many researchers (Castro and Liskov, 1999; Yin et al., 2003; Zhao, 2007b) and it is out of the scope of this paper.

For the purposes of checkpointing and recovery, we require that two additional operations, one for get_state and the other for set_state, be defined in the WSDL file for the Web services to be replicated. When it is needed to retrieve the application state, a get_state request message is delivered to the application and the reply should contain a snapshot of the application's current state. Likewise, when it is time to restore the application state according to a checkpoint, a set_state request message is delivered to the application.

We assume that all messages are protected by a digital signature to ensure their integrity and the adversaries have limited computing power so that they cannot break the digital signatures of correct replicas. Ideally, we could replace digital signatures by message authentication codes in most of cases to reduce the computational cost, and hence improve the runtime performance of our framework. However, because the Rampart library (an open-source implementation of the WS-Security standard) (Nadalin et al., 2004) that we use in our framework does not yet support

[3] The Apache Axis2 project, http://ws.apache.org/axis2/.

message authentication code, we decide to use digital signatures to protect all messages exchanged. The implementation of our framework can be upgraded relatively easily to using message authentication code when it becomes available in the future. For normal operation, the changes would be made mainly at the security handler, without disrupting other parts of the framework.

Finally, we assume that the clients who are interested in consuming the replicated Web services would negotiate a contract with the services provider and download all necessary software (including the client library for the BFT-WS framework) to get started. The initial locations of the service replicas provided to the client could be generic, and the requests from the client would then be dispatched (by a load balancer, for example) to the set of replicas designated for the client based on the client's identity and service contact. Dynamic service discovery, while an interesting topic, is out of the scope of this paper.

3.3. BFT-WS architecture

For increased flexibility, BFT-WS offers three different configurations, as shown in Fig. 3. The BFT-WS system architecture and the main components are first introduced for configuration I. The variations needed to accommodate the other two configurations are discussed subsequently. We conclude this section by discussing the support for advanced interaction patterns between Web services.

The configuration I follows the traditional BFT style where total ordering and execution of application requests are handled by the same set of $3f + 1$ replicas to tolerate up to f faulty replicas, and the configurations II and III are based on the idea of separation of agreement and execution (Yin et al., 2003). The difference between the latter two configurations is that in configuration II, $2f + 1$ of the $3f + 1$ agreement nodes are collocated with the $2f + 1$ execution nodes, and in configuration III, the $3f + 1$ agreement nodes are physically separated from the $2f + 1$ execution nodes, and therefore, they can be used to provide agreement service for multiple Web services.

3.3.1. Configuration I

The overview of the BFT-WS architecture for configuration I is shown in Fig. 4. BFT-WS is implemented as an Axis2 module. During the out-flow of a SOAP message, Axis2 invokes the BFT-WS Out Handler during the user phase, and invokes the Rampart handler for message signing during the security phase. Then, the message is passed to the HTTP transport sender to send to the target endpoint. During the in-flow of a SOAP message, Axis2 first invokes the default handler for preliminary processing (to find the target object for the message based on the URI and SOAP action specified in the message) during the transport phase, it then invokes the Rampart handler for signature verification during the security phase. This is followed by the invocation of the BFT-WS Global In Handler during the dispatch phase. This handler performs tasks

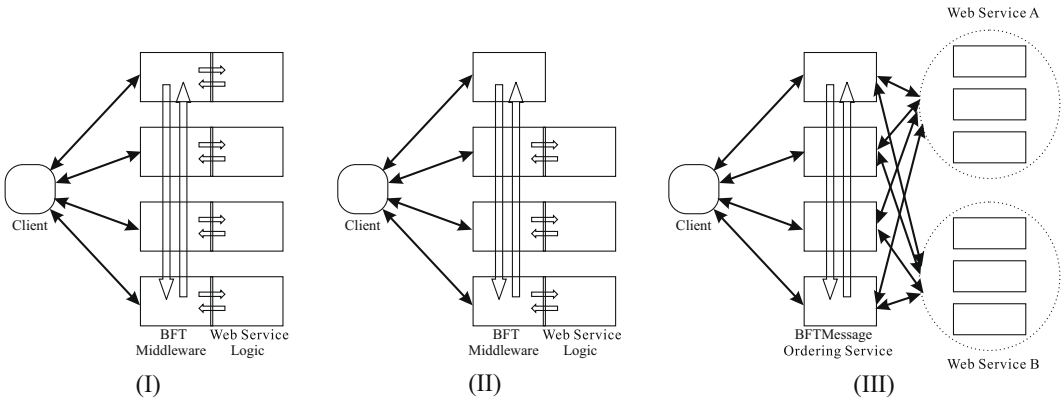

Fig. 3. Three alternative configurations of the BFT-WS framework.

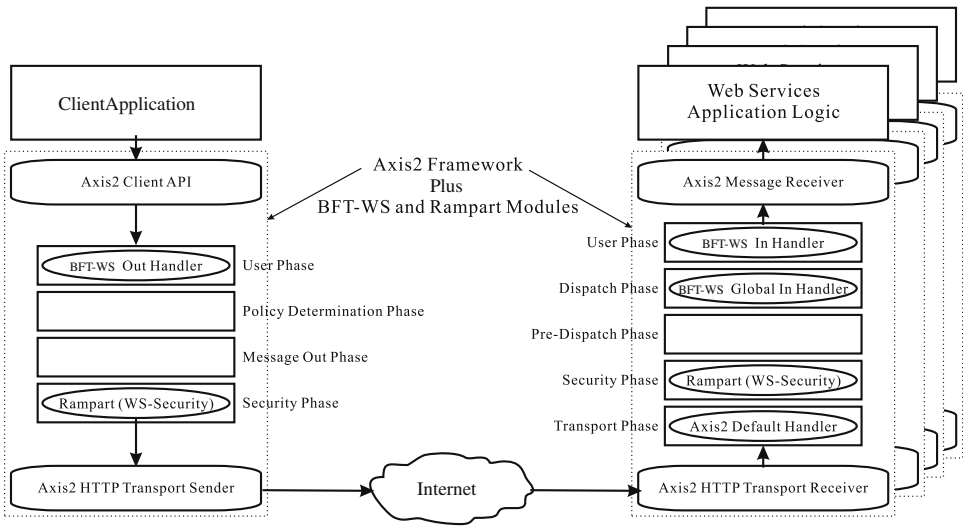

Fig. 4. The overview of the BFT-WS architecture in configuration I.

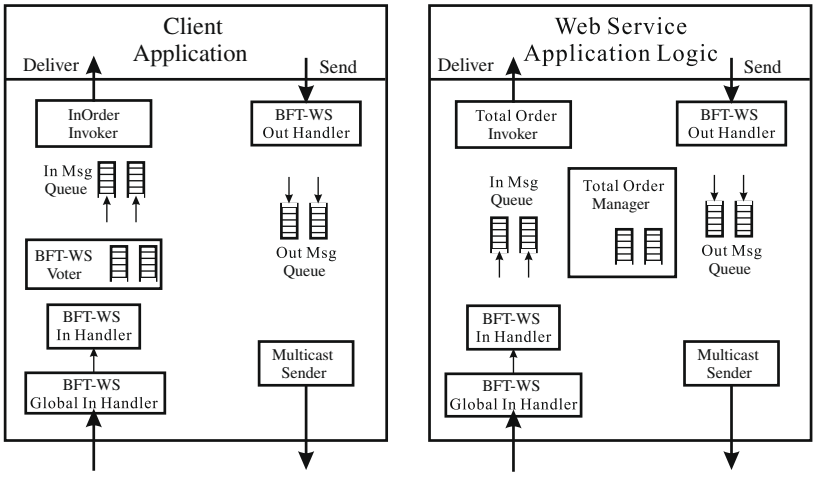

Fig. 5. The main components of the BFT-WS module.

that should be done prior to dispatching, such as duplicate suppression at the server-side. If the message is targeted toward a BFT-WS-enabled service, the BFT-WS In Handler is invoked for further processing during the user-defined phase, otherwise, the message is directly dispatched to the Axis2 message receiver. For clarity, Fig. 4 shows only a one-way flow of a request from the client to the replicated Web service. The response flow is similar. Also not shown in Fig. 4 are the multicast process and the internal components of the BFT-WS module.

The main components of the BFT-WS module are illustrated in Fig. 5. The client-side bears a lot of similarity to the Sandesha2 client-side module, with the exception of the addition of BFT-WS Voter, the replacement of Sandesha Sender by a Multicast Sender, and the replacement of the Sandesha Out Handler by the BFT-WS Out Handler. The server-side contains more additions and modifications to the Sandesha2 components. Furthermore, a set of actions are added to the module configuration to allow total ordering of messages, view change management and replica state synchronization. Besides the Multicast Sender, the server-side introduced a Total Order Manager, and replaced the original Global In Handler, In Handler, and InOrder handler, by BFT-WS Global In Handler, BFT-WS In Handler and Total Order Invoker, respectively. The storage framework in Sandesha2 is not changed. The functions of these components (both Sandesha2 original and the modified or new components) are elaborated in the following subsections, starting with the components dealing with the out-flow, and then the components for the in-flow. We note that the modification to Sandesha2 is carried out in a way such that BFT-WS is backward compatible with Sandesha2, *i.e.,* if BFT is not needed, the system can be reconfigured to support only WS-RM without the need of re-compilation or re-deployment of the Web services. Similarly, the BFT functionality can be turned on dynamically when the need occurs.

3.3.1.1. BFT-WS out handler. This handler performs out-flow processing for reliable messaging. In particular, it generates a CreateSequence request when the application sends the first message of a new sequence, and sends a terminate-sequence request after the last message of a sequence is transmitted.

The difference between the BFT-WS Out Handler and the original Sandesha Out Handler lies in the creation and handling of the CreateSequence message. In the original implementation, the CreateSequence message does not contain any element that can be used for the server-side to perform duplicate detection. If the CreateSequence request contains an Offer element, it may be used as a way to check for duplicate. However, not all CreateSequence requests contain such an element, because its existence is specified by the client application. To address this problem, we propose to include a UUID string in the CreateSequence request. The UUID is embedded in the CreateSequence/any element, an optional element specified by the WS-RM standard to enable extensibility.

The addition of this UUID element also helps alleviate a tricky problem that would cause replica inconsistency. The WS-RM standard does not specify how the sequence ID for the newly created sequence should be determined. In Sandesha2, a UUID string is generated and used as the sequence ID at the server-side. If we allow each replica to generate the sequence ID unilaterally in this fashion, the client would adopt the sequence ID present in the first CreateSequence response it receives. This would prevent the client from communicating with other replicas, and would prevent the replicas from referring to the same sequence consistently when ordering the application messages sent over this sequence. Therefore, we modified the CreateSequence request handling code to generate the sequence ID deterministically based on the client supplied UUID and the Web service group endpoint information.

3.3.1.2 Multicast sender. In BFT-WS, the sequence between the client and the service provider endpoints is mapped transparently to a virtual sequence between the client and the group of replicas. The same sequence ID is used for the virtual sequence so that other components can keep referring to this sequence regardless if it is a one-to-one or a one-to-many (or many-to-one) sequence. The mapping is carried out by the multicast sender.

To make the mapping possible, we assume that each service to be replicated bears a unique group endpoint, in addition to the specific endpoint for each replica. Higher level components, including the application, must use the group endpoint when referring to the replicated Web service. When a message to the group endpoint is detected, including application messages and BFT-WS control messages, the multicast sender translates the group endpoint to a list of individual endpoints and multicasts the message to these endpoints. We assume the mapping information is provided by a configuration file. The Multicast Sender runs as a separate thread and periodically poll the Out Message Queue for messages to send.

One additional change is the garbage collection mechanism. For point-to-point reliable communication, it is sufficient to discard a buffered message as soon as an acknowledgement for the message is received. However, this mechanism does not work for reliable multicast for apparent reasons. Consequently, a message to be multicast is kept in the buffer until the acknowledgement from all destinations have been collected, or a predefined retransmission limit has been exceeded.

Note that in BFT-WS, the client multicasts its requests to all replicas via the Multicast Sender component. Even though it may be less efficient in some scenarios, such as when the client is geographically farther away from the Web service and the Web service replicas are close to each other, this design is more robust against adversary attacks since the clients do not need to know which replica is currently serving as the primary. Without such information, the adversary can only randomly pick up a replica to attack, instead of focusing on the primary directly. From the availability perspective, the compromise of the primary can result in much severe performance degradation than that of a backup. It is important to encapsulate internal state information as much as possible to improve system robustness. Information encapsulation also reduces the dependency between the clients and the Web services.

3.3.1.3. BFT-WS global in handler. The Sandesha Global In Handler performs duplicate filtering on application messages. This is fine for the server-side, however, it would prevent the client from performing voting on the responses. Therefore, the related code is modified so that no duplicate detection is done on the client-side. The other functionalities of this handler, *e.g.,* generating acknowledgement for the dropped messages, is not changed.

3.3.1.4. BFT-WS in handler. Axis2 dispatches all application messages targeted to the BFT-WS-enabled services and the BFT-WS control messages to this handler. The BFT-WS In Handler operates differently for the client and the server-sides.

At the client-side, all application messages are passed immediately to the BFT-WS Voter component for processing. The rest of control messages are processed by the set of internal message processors as usual.

At the server side, all application messages are handled by an internal application message processor. Such messages are stored in the In Message Queue for ordering and delivery. All BFT-related control messages, such as pre-prepare, prepare, commit, and view change messages, are passed to the Total Order Manager for further processing. The WS-RM-related control messages such as CreateSequence and terminate-sequence requests, are handled by the internal message processors available from the original Sandesha2 module, with the exception of the handling of sequence ID creation.

3.3.1.5 BFT-WS voter. This component only exists at the client-side. The Voter verifies the authenticity of the application messages received and temporarily stores the verified messages in its data structure. For each request issued, the Voter waits until it has collected $f + 1$ identical response messages from different replicas before it invokes the application message handler to process the response message. When the processing is finished, the message is passed to the In Message Queue for delivery.

3.3.1.6. Storage manager. This component consists of the In Message Queue, the Out Message Queue, and a number of other subcomponents for sequence management, acknowledgement and retransmission management, and in-order delivery. This component comes with the Sandesha2 module. It is instrumented only for the purpose of performance profiling.

3.3.1.7. Total order invoker. This component replaces the Sandesha InOrder Invoker. This invoker runs as a separate thread to poll periodically the received application messages (stored in the In Message Queue) for ordering and delivery. To be eligible for ordering, the message must be in-order within its sequence, i.e., all previous messages in the sequence has been received and ordered (or being ordered). If the message is eligible for ordering, the Total Order Manager is notified to order the message. Note that only the primary initiates the ordering of application messages.

The Total Order Invoker asks the Total Order Manager for the next message to be delivered. If there is a message ready for delivery, the Invoker retrieves the message from the In Message Queue and delivers it to the Web service application logic via the Axis2 message receiver.

3.3.1.8. Total order manager. This component is responsible for imposing a total order on all application requests according to the BFT algorithm. To facilitate reliable communication among the replicas themselves, each replica establishes a sequence with the rest of the replicas. The reliability of the control messages sent over these sequences are guaranteed by the WS-RM mechanisms and the Multicast Sender. The Total Order Manager starts an instance of the BFT algorithm when a request that is in-order in its sequence becomes available, and it uses a TotalOrderBean object to keep track of the ordering status for each application message. A new TotalOrderBean is created when the first pre-prepare message for each application message is sent (at the primary) or accepted (at the backups). The Total Order Manager also maintain an OrderedMessages queue to store the ids of the application request messages that have already been totally ordered. An entry is removed from the OrderedMessages queue when the referenced message has been delivered to the application.

The Total Order Manager is also in charge of performing periodic checkpointing and garbage collections, and initiating/participating view changes according to the BFT algorithm. When performing a checkpointing, the manager injects a get_state request message at the *head* of the OrderedMessages queue and the message will be subsequently delivered to the application to produce a snapshot of the current application state. The retrieved state then is stored in a local data structure. When a new checkpoint becomes stable, the previous checkpoints, together with all control messages prior to the checkpoint, are garbage collected. During recovery, the recovering replica creates a set_state request message (with the checkpoint as the input parameter) and inserts it at the head of the OrderedMessages queue for delivery. When this message is dispatched, the application should restore its state according to the given checkpoint. State restoration might also be needed when a slow replica realizes that it has fallen too far behind (e.g., a message that it has missed is older than the latest stable checkpoint). In this case, the slow replica sends a fetch-state request to the primary for the latest stable checkpoint and the primary subsequently sends the entire stable checkpoint with proof (the set of $2f + 1$ checkpoint messages collected) to the requesting replica. Note that we choose not to follow the hierarchical state partitions based approach introduced in Castro and Liskov (1999) because the requirement for doing so is not compatible with the Java programming model. Our current approach could be optimized by considering application-specific state structure.

3.3.2. Configuration II

In Configuraiton II, the components described previously are virtually unchanged except that in Configuration II the Web services for the application are loaded only at $2f + 1$ replicas, i.e., the remaining f replicas do not deliver the application requests and they are not bothered with the handling of application responses. The major motivation for using this configuration is to save physical resources because the replicas that do not load the application Web services could be used to order requests for multiple different Web services, assuming that the throughput of these Web services are limited by application request processing rather than by the total ordering of requests.

3.3.3. Configuration III

Similar to Configuration II, this configuration also has the benefit of saving physical resources. In addition, by completely

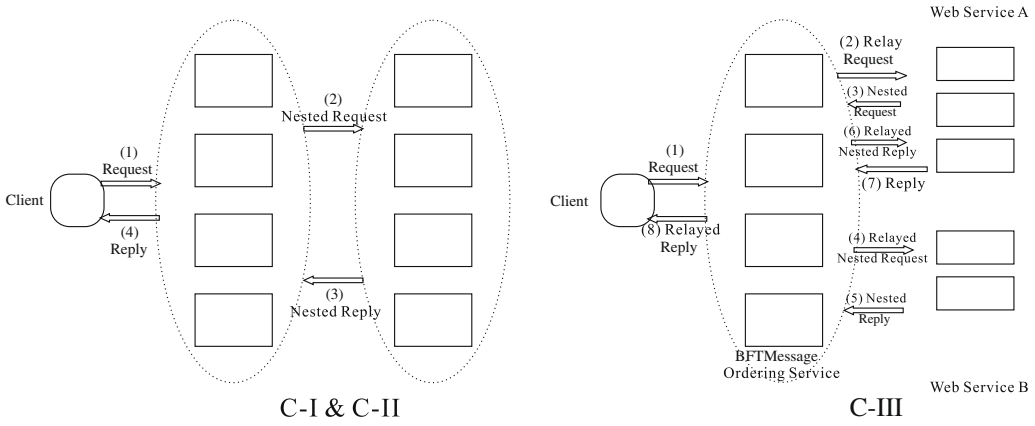

Fig. 6. Major communication steps (other than those for message ordering) for three-tier Web services interactions with different configurations.

separating the agreement nodes and the execution nodes, this configuration offers potentially better fault isolation and more flexibility in replication management. In Configurations I and II, since the execution of application requests is collocated with the agreement node, the compromise of the Web service logic due to a malformed application request, or a software bug in the application logic might disrupt the normal operation of the agreement process. The separation of the agreement nodes and the execution nodes enables the use of different replication degrees for the execution nodes and the agreement nodes according to the reliability analysis results of these two types of nodes.

3.4. Support for multi-tiered Web services interactions

Multi-tiered Web services interactions are common in practice. In such interactions, nested invocations are made by a Web service to another Web service, in response to an invocation. In essence, the Web service acts as the server for its clients and also as a client for some other Web service when issuing the nested invocations, as shown in Fig. 6.

In general, the client–server interaction implies a dependency relationship, i.e., the client depends on the services provided by the server, which usually imposes stronger dependability requirement on the server. In the context of multi-tiered Web services interactions, if a replicated Web service (in the middle-tier) invokes another Web service, it is logical to expect that the later is also replicated for fault tolerance. Even though BFT-WS allows a replicated Web service to invoke another, non-replicated Web service, this practice is discouraged because the use of a less dependable service would reduce one's own dependability.

Because the BFT-WS architecture described before contains modules for both client-side and server-side BFT mechanisms, extending it to support multi-tiered Web services interactions does not require major changes. Nevertheless, the following enhancements are needed.

A sequence between two replicated Web services will be mapped to a virtual sequence between two groups of replicas instead of between a single client and a group of replicas, as described in Section 3.3.1.2. The mechanism that enables the mapping is identical to that in Section 3.3.1.2. However, an interesting issue arises as to when a server replica should start to order a request sent by a replicated client. If the primary starts ordering a request as soon as it receives the message, the system would be vulnerable to the attack by a faulty client replica, e.g., if the faulty client replica sends a malformed nested request to all server replicas ahead of other (correct) client replicas' request, it would prevent the request from the correct client replicas from being accepted. Therefore, an additional filtering mechanism (similar to the voting mechanism for reply messages at the client-side) must be introduced. The filtering mechanism ensures that a server replica accepts a request from a replicated client if and only if it has collected $p+1$ identical requests from different client replicas, where p is the maximum faults tolerated by the replicated client. (The filtering mechanism should be used by a backend server even if it is not replicated.) For nested reply messages, the same client-side voting mechanism as described in Section 3.3.1.5 is employed.

4. Performance evaluation

Our performance evaluation is carried out on a testbed consisting of 20 Dell SC440 servers connected by a 100 Mbps Ethernet. Each server is equipped with a single Pentium D 2.8 GHz processors and 1GB memory running SuSE 10.2 Linux. In this section, we first present the experimental results to characterize runtime overhead of BFT-WS during normal operation, and then report the evaluation of checkpointing and recovery.

4.1. Runtime overhead characterization

An echo test application is used to characterize the runtime overhead. The client sends a request to the replicated Web service and waits for the corresponding reply within a loop without any "think" time between two consecutive calls. The request messages contains an XML document with varying number of elements, encoded using AXIOM (AXis Object Model).[4] At the replicated Web service, the request is parsed and a nearly identical reply XML document is returned to the client. For the multi-tier experiments, the echo test application is chained to the desirable number of tiers (i.e., in response to an echo request, the server invokes another echo server before sending out a reply). For the multi-tier experiment, the echo test application is modified such that an intermediate echo server relays the echo request to a backend echo server, and it would not respond to the request until it has received the response to the nested echo request.

In each run, 1000 samples are obtained. The end-to-end latency for the echo operation is measured at the client. The throughput are measured at the replicated Web service. In our experiment, we keep the number of replicas to 4 (for Configuration III, 4 replicas are used for agreement nodes and 3 replicas are used for Web services) to tolerate a single Byzantine faulty replica, and vary the

[4] Information regarding the Apache Axion can be found at http://ws.apache.org/commons/axion/.

request sizes in terms of the number of elements in each request, and the number of concurrent clients.

4.1.1. Client–server interactions

Fig. 7 shows the latency and throughput measurement results. In Fig. 7a, The end-to-end latency of the echo operation is reported for BFT replication with 4 replicas and a single client, for all three configurations (referred in the figure as C-I for Configuration I, C-II for Configuration II, and C-III for Configuration III). For comparison, the latency for two other configurations are also included. The first configuration involves no replication and no digital signing of messages. The second configuration involves no replication, but with all messages digitally signed. The measurements for these configurations reveal the cost of digital signing and verification. As can be seen, such cost ranges from 90 ms for short messages to 130 ms for longer messages. The latency overhead of running BFT replication is significant. However, the overhead is very reasonable considering the complexity of the BFT algorithm. Comparing with the latency for the no replication-with-signing configuration, the overhead for Configuration I ranges from 150 ms for short messages to over 310 ms for longer messages. The increased overhead for larger messages is likely due to the CPU contention for processing of the application requests (by the Web service) and the BFT replication mechanisms (by our framework). In Configuration II, the end-to-latency is slightly smaller than that in Configuration I because the primary is fully dedicated to message ordering and its operations are not subject to the CPU contention from the application processing. This effect is more prominent when the message complexity is higher. In Configuration III, the end-to-end latency is noticeably larger than Configurations I and II because the additional communication step introduced (i.e., the request and reply must be relayed between the agreement nodes and the execution nodes).

The throughput measurement results for different request sizes are shown in Fig. 7b–d. Note that the results for the no replication configurations include digital signing and verification of all messages for fair comparison. It can be seen from Fig. 7b that the throughput degradation is about 50% when BFT replication (Configuration I) is enabled for short request sizes. Again, this is anticipated. In Configuration I, even with optimal batching for 8 concurrent clients, the primary must multicast 2 control messages (pre-prepare and commit) and receive 6 control messages (3 prepare and 3 commit messages from backups) to order the 8 application requests (recall that our measurements are carried out for normal operation with no replica failure). The approximately 50% reduction in throughput for short messages is nearly optimal. When the application request length and complexity is increased, the throughput reduction becomes far less, as shown in Fig. 7c and d. In Configuration II, the throughput is slightly higher than that for Configuration I, again likely due to the less CPU contention. In Configuration III, the throughput is lower than that for Configuration I because of two reasons: (1) the extra communication step incurs additional CPU cycles for sending/receiving, and for digital signature verification, and (2) due to the larger end-to-end latency in Configuration III, the request arrival rate under the same number of concurrent clients is lower than that in Configurations I and II.

4.1.2. Multi-tier interactions

The results of the performance measurements for a three-tier interaction between a set of clients (up to eight) and two Web ser-

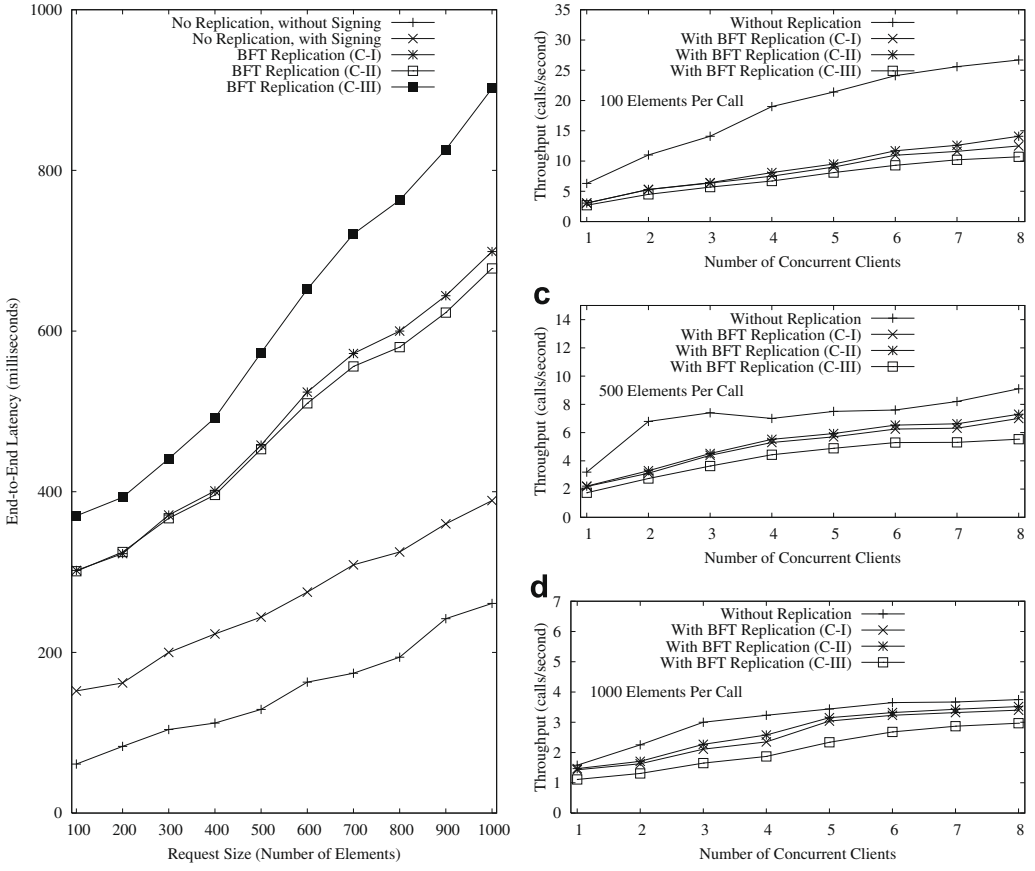

Fig. 7. BFT-WS performance during normal operation. (a) The end-to-end latency. For comparison, the latency for the no-replication configuration with and without digital signing of messages are included as well. (b)–(d) Throughput vs. number of concurrent clients with different message sizes.

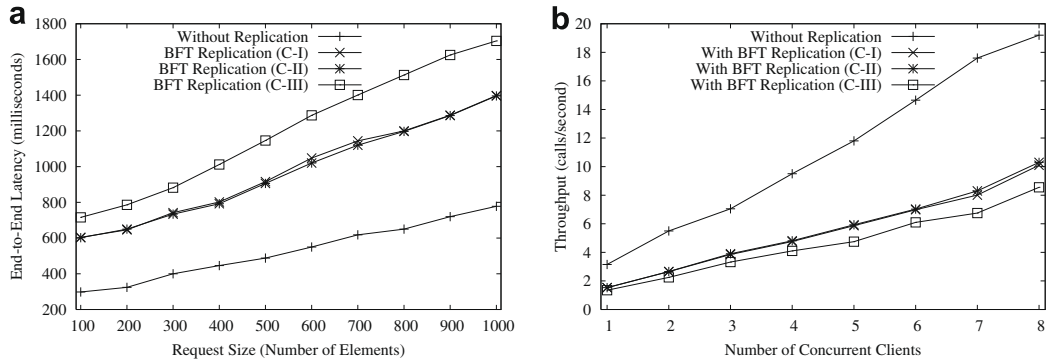

Fig. 8. BFT-WS performance for a three-tier echo application. (a) The end-to-end latency. (b) Throughput vs. number of concurrent clients with a request size of 100 elements.

vices under different configurations are shown in Fig. 8. As can be seen in Fig. 8a, the end-to-end latency virtually doubled that of the client–server experiment. This is expected due to the extra communication steps as illustrated in Fig. 6. Correspondingly, the throughput observed in Fig. 8b is much smaller than that of the client–server experiment, under the same number of concurrent clients because the request arrival rate is virtually half of the value in the client–server experiment.

4.2. Evaluation of checkpointing and recovery

For all the experiments in this section, only Configuration I is used, and the number replicas is kept at 4.

4.2.1. Checkpointing

For the purpose of garbage collection and recovery, each replica periodically performs checkpointing of its state. The cost of taking a checkpointing is application-dependent because it is dominated by the size and complexity of the application state. In our experiment, the application state consists of multiple XML elements, and its size (in terms of the number of XML elements) is provided via command-line argument when the replicas are started (i.e., the state size is fixed during the life-cycle of each replica for the sake of our measurements). The measurements are carried out in the presence of a single client that continuously issues requests to the replicas. A checkpoint is taken for every 10 requests ordered and executed. During these measurements, no fault is introduced.

The checkpointing latency, i.e., the time it takes for a replica to obtain a checkpoint, for different state sizes (in terms of the number of elements) is shown in Fig. 9a. Also shown in Fig. 9b is the measured average time for a checkpoint to become stable (i.e., when $2f + 1 = 3$ checkpoint messages for the same sequence has been received) since a replica multicasts a checkpoint message.

4.2.2. State transfer

Because it rarely happens for the replica to run so slowly that it requires a state transfer to catch up with other replicas, we measure the state transfer time by restarting a replica with the presence of a single client (that continuous issues requests to the replicas). When the replica restarts, it would send a fetch-state request to the primary, wait for the latest stable checkpoint with proof, and restore its state accordingly. The state transfer latency is the time from the sending of the fetch-state request until the replica restores its state. The measurement results are summarized in Fig. 10. The state transfer latency is dominated by the time it takes for the primary to transmit its latest stable checkpoint with proof to the recovering replica, and the time it takes for the recovering replica to restore its state, both exhibiting strong dependency on the checkpoint size.

4.2.3. View change

In this experiment, we introduce view changes by shutting down the primary of the current view and then measure the view change latency at the new primary for the next view in the presence of a single client. The view change latency is the time between the sending of a view-change request and the installation of the new view (i.e., the sending of the new-view message after the new primary has collected $2f + 1 = 3$ view-change requests and completed the calculation for the new view).

Since the replicas are running on identical hardware, we do not observe message losses and no state-transfer is triggered in this experiment. Furthermore, the view change induced by shutting down the primary always succeeds, i.e., the primary in the new view is non-faulty and our testbed offers sufficient synchrony for the view change to complete within the preset view change timeout of 2 s.

As shown in Fig. 11, the view change latency has strong dependency on the checkpoint interval. This is because a large check-

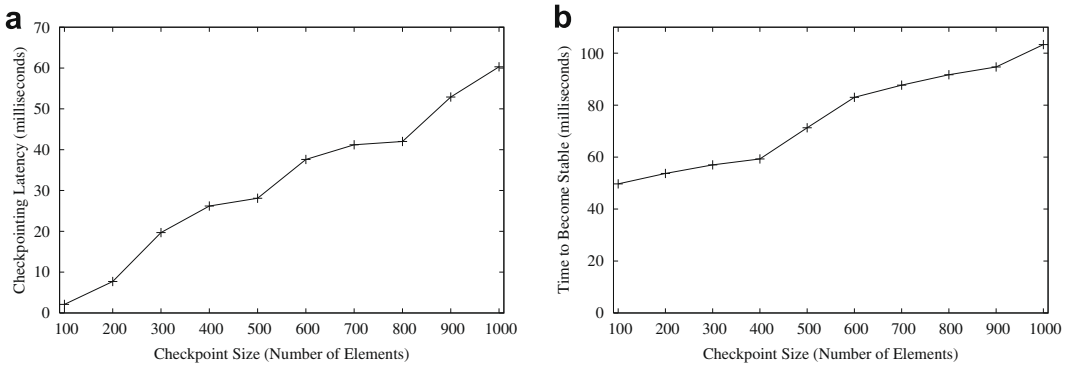

Fig. 9. (a) Checkpointing latency with respect to the size of the state. (b) The average time for a checkpoint to become stable.

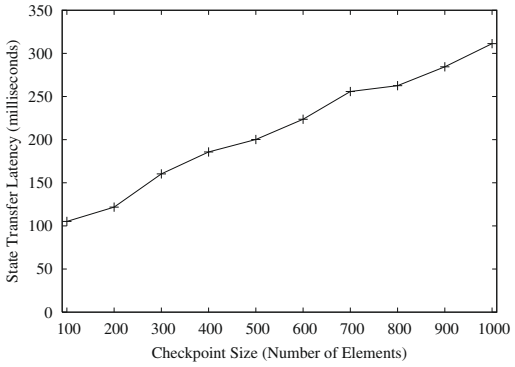

Fig. 10. State transfer latency.

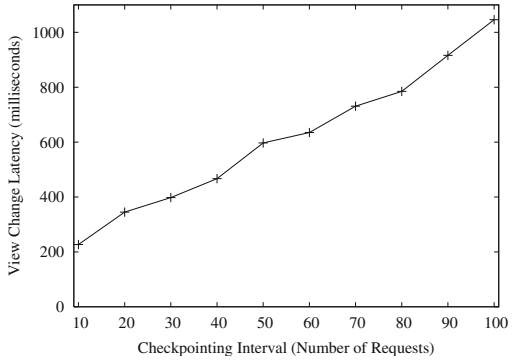

Fig. 11. View change latency.

point interval often leads to the inclusion of the pre-prepare and prepare records for many requests (the view change request must contain records for each request ordered after the latest stable checkpoint). This observation reveals that in practical deployments of BFT frameworks, we must select an appropriate checkpoint interval so that a good balance can be struck between achieving good normal operation performance and ensuring relatively short fail-over latencies.

5. Related work

A large number of high availability solutions for Web services have been proposed in the last several years (Birman, 2004; Chan et al., 2006; Dialani et al., 2002; Dobson, 2006; Erradi and Maheshwari, 2005; Fang et al., 2007; Looker et al., 2005; Merideth et al., 2005; Moser et al., 2006; Pallemulle et al., 2008). Most of them are designed to cope with crash faults only and very few offered Byzantine fault tolerance capability (Merideth et al., 2005, 2008, 2005). While SWS (Li et al., 2005) proposed a few protocols to achieve Byzantine fault tolerance for Web services, no prototype system is implemented and only simulation results are reported.

Similar to our work, Thema (Merideth et al., 2005) also relies on the BFT algorithm to ensure total ordering of application messages. However, a wrapper is used to interface with an existing implementation of the BFT algorithm (Castro and Liskov, 1999), which is based on IP multicast, rather than the standard SOAP/HTTP transport, as such, it suffers from the interoperability problem we mentioned in the beginning of this paper. This approach limits its practicality.

Very recently, we became aware of Perpetual-WS (Pallemulle et al., 2008), which is another BFT framework for Web services. It is also implemented on top of Axis2 by exploiting Axis2's pluggability. Therefore, similar to our BFT-WS framework, it can be categorized as the integrated approach. (Perpetual-WS offers a set of proprietary APIs for Axis2 clients to enable asynchronous communication. Presumably, such APIs could be dropped when Axis2 adds such feature in the future.) Furthermore, Perpetual-WS supports asynchronous communication for multi-tiered applications, which could offer much better throughput than BFT-WS. However, Perpetual-WS does not support WS-RM, which may limit its adoption for practical systems.

6. Conclusion

In this paper, we presented the design and implementation of BFT-WS, a Byzantine fault tolerance middleware framework for Web services. It uses standard Web services technology to build the Byzantine fault tolerance service, and hence, it is more suitable to achieve interoperability. Furthermore, BFT-WS is backward compatible with WS-RM. BFT-WS is carefully designed and implemented so that when there is no need to replicate a Web service, a single instance of the Web service can run with the default WS-RM implementation instead of our BFT mechanisms. We also documented in detail the architecture and the major components of our framework. We anticipate that such descriptions are useful to practitioners as well as researchers working in the field of highly dependable Web services. Finally, our framework has been carefully tuned to exhibit optimal performance, as shown in our performance evaluation results.

References

Abd-El-Malek, M., Ganger, G.R., Goodson, G.R., Reiter, M., Wylie, J.J., 2005. Fault-scalable Byzantine fault-tolerant services. In: Proceedings of ACM Symposium on Operating System Principles.

Bilorusets, R. et al., 2005. Web Services Reliable Messaging Specification. URL <http://www.ibm.com/developerworks/library/specification/ws-rm>.

Birman, K., 2004. Adding high availability and autonomic behavior to Web services. In: Proceedings of the 26th International Conference on Software Engineering, Scotland, UK.

Castro, M., Liskov, B., 1999. Practical Byzantine fault tolerance. In: Proceedings of the Third Symposium on Operating Systems Design and Implementation, New Orleans, USA.

Castro, M., Rodrigues, R., Liskov, B., 2003. BASE: using abstraction to improve fault tolerance. ACM Transactions on Computer Systems 21 (3), 236–269.

Chan, P., Lyu, M., Malek, M., 2006. Making services fault tolerant. Lecture Notes in Computer Science 4328, 43–61.

Chen, L., Avizienis, A., 1995. N-version programming: a fault-tolerance approach to reliability of software operation. In: Proceedings of the 25th International Symposium on Fault-Tolerant Computing, p. 113.

Cowling, J., Myers, D., Liskov, B., Rodrigues, R., Shrira, L., 2006. HQreplication: a hybrid quorum protocol for Byzantine fault tolerance. In: Proceedings of the Seventh Symposium on Operating Systems Design and Implementations, Seattle, Washington.

Dialani, V., Miles, S., Moreau, L., Roure, D.D., Luck, M., 2002. Transparent fault tolerance for web services based architecture. Lecture Notes in Computer Science 2400, 889–898.

Dobson, G., 2006. Using WS-BPEL to implement software fault tolerance for Web services. In: Proceedings of the 32nd EUROMICRO Conference on Software Engineering and Advanced Applications.

Erradi, A., Maheshwari, P., 2005. A broker-based approach for improving Web services reliability. In: Proceedings of the IEEE International Conference on Web Services, Orlando, Florida.

Fang, C., Liang, D., Lin, F., Lin, C., 2007. Fault tolerant web services. Journal of Systems Architecture 53, 21–38.

Kotla, R., Alvisi, L., Dahlin, M., Clement, A., Wong, E., 2007. Zyzzyva: speculative Byzantine fault tolerance. In: Proceedings of ACM Symposium on Operating System Principles.

Lamport, L., Shostak, R., Pease, M., 1982. The byzantine generals problem. ACM Transactions on Programming Languages and Systems 4 (3), 382–401.

Li, W., He, J., Ma, Q., Yen, I.-L., Bastani, F., Paul, R., 2005. A framework to support survivable web services. In: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium, p. 93102.

Looker, N., Munro, M., Xu, J., 2005. Increasing web service dependability through consensus voting. In: Proceedings of the 29th Annual International Computer Software and Applications Conference. pp. 66–69.

Merideth, M., Iyengar, A., Mikalsen, T., Tai, S., Rouvellou, I., Narasimhan, P., 2005. Thema: Byzantine-fault-tolerant middleware for web services applications. In: Proceedings of the IEEE Symposium on Reliable Distributed Systems, pp. 131–142.

Moser, L., Melliar-Smith, M., Zhao, W., 2006. Making web services dependable. In: Proceedings of the First International Conference on Availability, Reliability and Security, Vienna University of Technology, Austria, pp. 440–448.
Nadalin, A., Kaler, C., Hallam-Baker, P., Monzillo, R., 2004. Web Services Security: SOAP Message Security 1.0. OASIS, Oasis Standard 200401 Edition.
Pallemulle, S.L., Thorvaldsson, H.D., Goldman, K.J., 2008. Byzantine fault-tolerant Web services for n-tier and service oriented architectures. In: Proceedings of the 28th International Conference on Distributed Computing Systems, pp. 260–268.
Salas, J., Perez-Sorrosal, F., Patino-Martinez, M., Jimenez-Peris, R., 2006. Ws-replication: a framework for highly available web services. In: Proceedings of the 15th International Conference on World Wide Web, Edinburgh, Scotland, pp. 357–366.
Schneider, F., 1990. Implementing fault-tolerant services using the state machine approach: a tutorial. ACM Computing Surveys 22 (4), 299–319.
Yin, J., Martin, J.-P., Venkataramani, A., Alvisi, L., Dahlin, M., 2003. Separating agreement from execution for byzantine fault tolerant services. In: Proceedings of the ACM Symposium on Operating Systems Principles. Bolton Landing, NY, USA, pp. 253–267.
Zhao, W., 2007a. BFT-WS: a Byzantine fault tolerance framework for Web services. In: Proceedings of the Middleware for Web Services Workshop, Annapolis, MD, USA, pp. 89–96.
Zhao, W., 2007b. Byzantine fault tolerance for nondeterministic applications. In: Proceedings of the Third IEEE International Symposium on Dependable, Autonomic and Secure Computing. Loyola College Graduate Center, Columbia, MD, USA, pp. 108–115.
Zhao, W., Moser, L.E., Melliar-Smith, P.M., 2004. Design and implementation of a pluggable fault tolerant CORBA infrastructure. Cluster Computing: The Journal of Networks, Software Tools and Applications Special issue on Dependable Distributed Systems 7 (4), 317–330.