Byzantine fault tolerance for session-oriented multi-tiered applications

Hua Chai and Wenbing Zhao*

Department of Electrical and Computer Engineering,
Cleveland State University,
Cleveland, OH 44115, USA
E-mail: h.chai@csuohio.edu
E-mail: w.zhao1@csuohio.edu
*Corresponding author

Abstract: This article presents a lightweight Byzantine fault tolerance (BFT) framework for session-oriented multi-tiered applications. We conclude that it is sufficient to use a lightweight BFT algorithm instead of a traditional BFT algorithm, based on a comprehensive study of the threat model to, and the state model of, the session-oriented multi-tiered applications. The lightweight BFT algorithm uses source ordering, rather than total ordering, of incoming requests to achieve Byzantine fault tolerant state-machine replication of such type of applications. The performance of the lightweight BFT framework is evaluated using a shopping cart application prototype built on the web services platform. The same shopping cart application is used as a running example to illustrate the problem we address and our proposed solution. Performance evaluation results obtained from the prototype implementation show that indeed our lightweight BFT algorithm incurs very insignificant overhead.

Keywords: Byzantine fault tolerance; BFT; multi-tiered applications; web services; service-oriented computing; trustworthy computing.

Reference to this paper should be made as follows: Chai, H. and Zhao, W. (2013) 'Byzantine fault tolerance for session-oriented multi-tiered applications', *Int. J. Web Science*, Vol. 2, Nos. 1/2, pp.113–125.

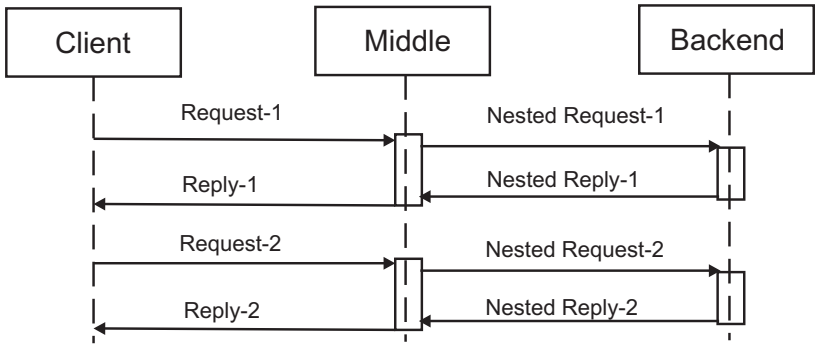
Biographical notes: Hua Chai is a doctoral student in the Department of Electrical and Computer Engineering at Cleveland State University (CSU). She received her Master of Science in Electrical Engineering in 2009 from CSU. Her research interests include fault tolerance computing, event streaming processing, and service-oriented computing.

Wenbing Zhao received his PhD in Electrical and Computer Engineering from the University of California, Santa Barbara, in 2002. He is an Associate Professor in the Department of Electrical and Computer Engineering at Cleveland State University. His current research interests include distributed systems, computer networks, fault tolerance and security. He has more than 80 academic publications.

1 Introduction

The multi-tiered architecture has been the predominant architecture for web-based applications because it facilitates the separation of business logic execution and (persistent) state management, which is the foundation to scale a web application to the internet scale. In the multi-tiered architecture, as shown in Figure 1, the client tier is responsible for the presentation of the web services via a graphic user interface, one or more middle-tier servers are tasked to process the requests issued by the client tier according to pre-defined business logic, and the backend servers maintain persistent data for the application.

Figure 1 Typical interactions in a three tier application



A user often interacts with a web service (constructed using the multi-tiered architecture) within the boundary of a session. The session is typically initiated when the user first logs in to his/her account and ends when the user explicitly terminates the session by logging out or when the session is timed out (when the user abandons the session). Session-oriented computing is pervasive in web-based applications because it fits well with the user interaction semantics and works nicely with the state management needs. Typically, the middle-tier server fetches data related to the user from the backend server at the initiation of a new session and such data is maintained at the middle-tier server in the form of session state.

Considering the untrusted operating environment of the internet, the middle-tier servers are vulnerable to many forms of threats and how well the middle-tier servers are protected against such threats are crucial to the trustworthiness of the *entire* web service. For example:

- Due to a crash or malicious fault, the middle-tier server may fail to respond to the client's request. If the middle-tier server is offering a shopping cart service to users, the users would not be able to make a purchase, which would result in the loss of business opportunities.

- If the middle-tier server is compromised, the integrity of the service can no longer be guaranteed, e.g., for a shopping cart service, the type and quantity of the product ordered by the client may be altered.

Byzantine fault tolerance (BFT) (Castro and Liskov, 2002; Clement et al., 2009a) is a promising technology that could help an application achieve high availability and trustworthiness. A Byzantine fault (Lamport et al., 1982) refers to an arbitrary fault, which could be a crash or malicious fault. BFT can be achieved by using space redundancy and ensuring that all non-faulty replicas reach an agreement, referred to as Byzantine agreement, on the total ordering of requests to the replicated server despite the presence of Byzantine faulty replicas and clients.

Naturally, one can use an existing BFT algorithm, such as PBFT (Castro and Liskov, 2002), to enhance the trustworthiness of the middle-tier servers to control the example threats to the system described above. However, such BFT algorithms are designed for generic stateful applications and often incur significant runtime overhead. We recognise that many web-based applications are designed to be session-oriented and each session involves only a single client. This observation motivates us to look for a much lighter weight solution to the problem where maintaining the objective of high trustworthiness of such applications. We conclude that a lightweight BFT algorithm can be designed by exploiting the semantics of the applications. In this article, we present such as a study. A lightweight BFT algorithm is developed based on the insight obtained via a comprehensive analysis of the state model and the threat model on session-oriented multi-tiered applications.

We have implemented a BFT framework running our lightweight BFT algorithm. A shopping cart application is used both as a running example to illustrate our approach and for performance evaluation of our framework. First, we analyse the threats to the multi-tiered architecture shown in Figure 1, using the shopping cart application as an example, and explore strategies to mitigate such threads. Second, we analyse the state model of the application. The analysis reveals that a source ordering (instead of total ordering) of application requests is sufficient, which is the basis for our lightweight BFT algorithm. The description of the lightweight algorithm as well as its proof of correctness are presented. The performance of the algorithm is carefully evaluated and compared against the performance of non-replicated application. The results show that our lightweight BFT algorithm incurs a very modest runtime overhead.

2 The shopping cart application

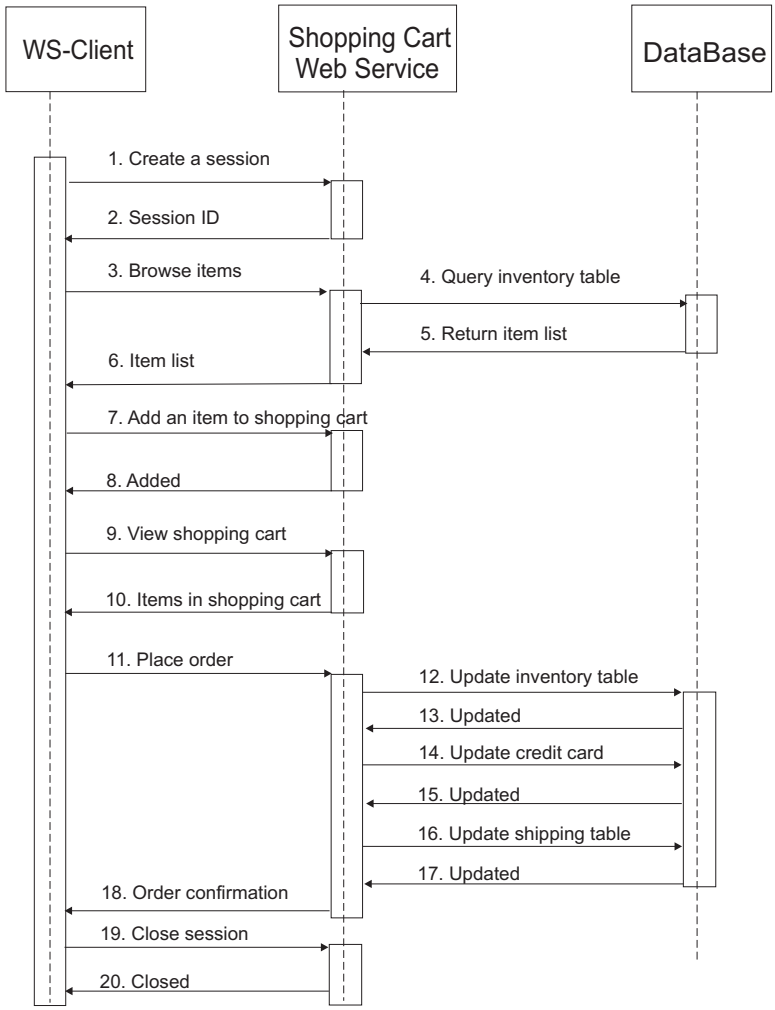
Throughout this article, a shopping cart application shown in Figure 2 is used as an example to illustrate the problem we address and our proposed solution for session-oriented multi-tiered web services applications.

When a client first invokes the shopping cart web service, the service instance at the middle-tier creates a session and a shopping cart object for this client. All messages exchanged within this session will carry a unique session identifier. With the session identifier as a reference, the client can operate on its shopping cart with a sequence of operations through the shopping cart web service. To process an invocation from the client, the shopping cart web service may issue nested invocations on the backend database server. When the session is over, the shopping cart web service closes the session for the client and removes the data stored for the session.

The creation of a session is triggered by the client's first invocation to the shopping cart web service (step 1). After a session gets created, the shopping cart web service returns a session identifier to the client (step 2). With the session identifier as a reference,

the client can carry out a sequence of operations to manage its shopping cart. As shown in Figure 2, the client requests to browse items (step 3) and then the web service issues a nested invocation to the backend database server to obtain item information (step 4). After getting a reply back from the database, the web service composes a list of items and sends the information to the client (step 5 and step 6). Once receiving the list of items, the client requests to add one of the items to its shopping cart and receives a reply (step 7 and step 8). If it successfully adds the item, the client checks its shopping cart and reviews the added item in its cart (step 9 and step 10). The client then places an order (step 11), and the web service initiates a transaction with three updates to the backend database server (steps 12–17). After the transaction is committed, the web service returns the order confirmation to the client (step 18). The client requests to close the session after it receives the order confirmation (step 19) and the web service notifies the client that the session is closed and purges the session state for the client (step 20).

Figure 2 Web service shopping cart example



2.1 Threat analysis

In this section, we analyse the threats that can potentially compromise the integrity of the middle-tier service.

2.1.1 Threats from a faulty middle-tier server

A faulty middle-tier server could:

1 Refuse to respond to the client's request.

2 Lie about its execution status and send replies that are inconsistent with the state stored at the backend database server to the clients. For example, the middle-tier server for the shopping cart service could report that the order is successfully placed when in fact it failed.

3 Lie about its execution status and insert spurious data into or modify correct data maintained at the backend database server. For example, the middle-tier server could modify the client's order by adding product into the order that the client did not order.

In case 1, the service is rendered unavailable to the client. This threat can be controlled by replicating the middle-tier server, which is automatically addressed by our lightweight BFT algorithm. In cases 2 and 3, the integrity of the service is compromised, e.g., the web service could have the credit card of the client charged without inserting a shipping record to the backend database. The web service could fail to execute the request to place an order but still report to the client that the request has been successfully executed. The web service could also insert shipping records for items that none of the clients has purchased. To control the threats in 2 and 3, in addition to replicating the middle-tier server, a Byzantine agreement is necessary on requests sent to server replicas and their relative ordering, and furthermore, the replies received at the client and the nested requests at the backend database server must also be voted on, which can be accomplished by using any of the well-known BFT algorithms. As to be discussed later, we aim to derive a much lighter weight solution to achieve the same objective.

2.1.2 Threats from a faulty client

A faulty client could

1 Send malformed requests to the middle-tier server.

2 Send conflicting requests to different replicas of the middle-tier server. For example, the faulty client in the shopping cart application could request different products to be added into the shopping cart at different replicas, or placing an order at some replicas while clearing the shopping cart at other replicas.

The threats in case 1 cannot be addressed by any BFT algorithm because such algorithms only ensure the consistency of the replicas and not the validity of the requests. How to address the validity of the requests is out of the scope of this article.

The threat in case 2 can be controlled by replicating the middle-tier server and by ensuring a Byzantine agreement on the requests and their relative ordering to the server replicas. However, because a faulty client could corrupt the state associated with the

session anyway (as in case 1), and other sessions will not be directly impacted, we do not see any benefit of controlling this threat.

2.2 State model analysis

The state model for session-oriented applications has the following characteristics:

1 The states for different sessions at the middle-tier server are disjoint.

2 Different sessions may indeed share state, but only through the backend server. The execution order of requests (to the middle tier) with conflicting nested invocations is naturally synchronised at the backend server.

The above observation implies that:

1 operations belonging to different sessions can be processed in parallel and their relative ordering does not matter

2 it is sufficient to ensure that the requests within the same session are delivered to different replicas of the middle-tier server in the same order.

Because each session is involved with a single client, such as the shopping cart application, source ordering, instead of total ordering, of the requests within each session is adequate to ensure replica consistency.

3 The lightweight BFT algorithm

In this section, we present the lightweight BFT algorithm and the proof of its correctness.

3.1 Assumptions

We assume that the messages can be reliably exchanged between different entities in a session-oriented multi-tiered application, which can be trivially satisfied by using TCP (in the transport level) and web services reliable messaging [i.e., *WS-ReliableMessaging* (Davis et al., 2008)] (in the application level). In particular, if a client or the database server sends a message to a non-faulty replica, the replica will eventually receive the message. We also assume that there are $2f + 1$ replicas for the middle-tier server, of which at most f are faulty. Each replica has a unique identifier number k, where k ranges from 0 to $2f$. All replicas play an equal role (i.e., no one is designated as the primary). We assume that the backend database server is trusted (e.g., can be achieved by using BFT replication).

We further assume that

1 All messages exchanged are uniquely identified (i.e., each message carries a unique identifier), which can be achieved by using web services facilities [such as Apache Axis2 (http://ws.apache.org/axis2/)].

2 All messages sent from replicas are all digitally signed (or protected with message authentication code as an optimisation), which can be satisfied by using WS-security [such as Apache *WSS4J* (http://ws.apache.org/wss4j/)] or Java security.

3.2 Properties

Our lightweight BFT algorithm for the trustworthy session-oriented middle-tier service satisfies the following two properties:

P1 The faulty replicas (of the middle-tier server) cannot effect unauthorised changes at the backend server.

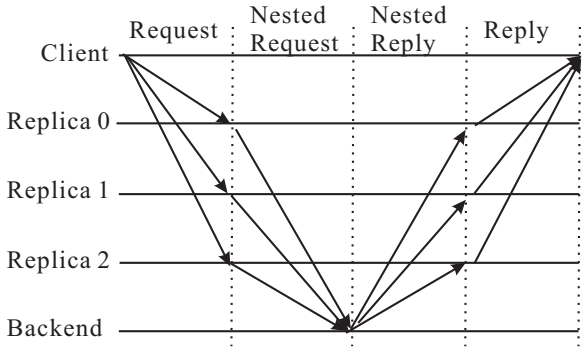
P2 Concurrent processing for different sessions at the middle-tier server will not cause inconsistent views of persistent data at different non-faulty replicas.

P3 If a non-faulty client sends a request to the middle-tier server, it will receive a correct reply and any state changes will be effected correctly at the backend server.

3.3 The lightweight BFT algorithm

The operation of the algorithm for a multi-tier application (such as the shopping cart application) with $f = 1$ is shown in Figure 3.

Figure 3 The lightweight BFT algorithm



A client sends the request to all replicas (of the middle-tier server). The request has the form < CLIENT-REQUEST t, m, c $>_{\sigma_c}$, where t is the timestamp, m is the message context, which consists of the operation requested together with all the necessary parameters, c is the client identifier, and σ_c is the digital signature of the request message m signed by the client c (as an optimisation, the message authentication code can be used in lieu of digital signature). The tuple < t, c > forms the message identifier of the request.

On receiving a request, a middle-tier replica validates the signature σ_c. If the request can be verified for m, and no other request that carries the same message identifier has been accepted, the request is accepted and executed. The execution may involve nested invocations on the backend server and in this case, the replica composes a database request < DB-REQUEST s, n, *sql*, r $>_{\sigma_r}$ to send to the backend server, where s is the session identifier, n is the sequence number (indicates the number of nested invocations issued within a session), *sql* is the SQL statement, r is the replica identifier, and σ_r is the digital signature of the replica.

If the backend server receives at lease $f + 1$ verifiable consistent nested requests from different replicas and it has not accepted a request with the same s and n, the backend server accepts the request, executes the SQL statement *sql* and returns a reply to all

replicas. The reply from the backend server has the following form < DB-REPLY s, n, s_r $>_{\sigma_{db}}$, where s_r consists of the response to the SQL query or update, and σ_{db} is the digital signature of the backend server. A replica accepts a nested reply when it contains the same s and n. It is possible for a replica to receive such a nested reply before it has issued the corresponding nested request, in which case, the nested reply is queued at the replica until the corresponding nested request is issued.

When a replica finishes processing and completes all nested invocations, it replies to the client < REPLICA-REPLY t, m, c $>\sigma_r$. If the client can collect $f + 1$ verifiable replies, with the same message identifier and the same message context as the request it has sent earlier, from different replicas, it accepts the reply and delivers it to the application.

It is worth noting that to avoid generating the session identifier non-deterministically, we use the message identifier of the first invocation of a client to determine the session identifier.

3.4 Proof of correctness

We provide an informal proof of correctness of the lightweight BFT algorithm in terms of the properties given in Section 3.1.

Theorem 1: The faulty replicas (of the middle-tier server) cannot effect unauthorised changes at the backend server.

Proof: Because the backend server does not accept a nested request unless it has collected $f + 1$ consistent requests, and there are at most f faulty replicas, it is obvious that they cannot effect any authorised changes at the backend server.

Theorem 2: Concurrent processing for different sessions at the middle-tier server will not cause inconsistent views of persistent data at different non-faulty replicas within the same session.

Proof: Persistent data is managed by the backend server. Concurrent processing for different sessions at the middle-tier server will naturally be serialised by the backend server if such processing involves conflicting operations on shared persistent data. Hence, no two non-faulty replicas within the same session could possibly see different values of the same shared persistent data.

Theorem 3: If a non-faulty client sends a request to the middle-tier server, it will receive a correct reply and any state changes will be effected correctly at the backend server.

Proof: We first prove by contradiction on the claim that any state changes will be effected correctly at the backend server. If the state changes are not effected correctly at the backend server, we consider the following three possible scenarios:

1 it is due to a malformed request that is inconsistent with the client's intention and/or business logic, for example, an order submission with altered quantity or product in a shopping cart application

2 it is due to the acceptance (and processing) of duplicate requests, such as duplicate order submission in a shopping cart application

3 it is due to the absence of a nested request that should have been accepted at the backend server.

In scenario 1, it implies that such a malformed nested request has been accepted at the backend server, which means at least $f + 1$ replicas have issued such a request. This is impossible because there are only up to f faulty replicas. (By definition, a non-faulty replica does not issue malformed nested requests.)

In scenario 2, each of the duplicate nested requests must have been accepted at the backend server, which means that at least $f + 1$ replicas have issued each of them and each of them carries a different message identifier. This implies that at least one non-faulty replica has issued duplicate message with different message identifiers, which is impossible.

In scenario 3, the absence of a nested request implies that no more than f replicas have issued the nested request, which is impossible because the request issued by a non-faulty client will reach all non-faulty replicas and there are at least $f + 1$ of them.

The correct state changes at the backend server, together with correct processing at the $f + 1$ non-faulty replicas, ensure that the request is properly handled and a correct reply is sent to the client. Because there are $f + 1$ or more non-faulty replicas, it is guaranteed that the client can receive at least $f + 1$ consistent replies, which ensures the delivery of the reply.

4 Performance evaluation

The implementation of the lightweight BFT framework and the shopping cart example application is based on a set of Apache web services facilities, including Apache Axis2 (http://ws.apache.org/axis2/) and Apache WSS4J (http://ws.apache.org/wss4j/), and Apache Derby (http://db.apache.org/derby/), a Java-based open-source database system. WSS4J is an implementation of the WS-Security standard (Nadalin et al., 2006). Most of the mechanisms are implemented as Axis2 handlers or modules which can be easily plugged into the framework without affecting other components. We use HMAC supported by WSS4J for message protection.

The performance of our lightweight BFT algorithm is evaluated in a local-area network (LAN) testbed that consists of 14 HP BL460c blade servers connected by a Cisco 3020 Gigabit switch. Each blade server is equipped with two Xeon E5405 (2 GHz) processors and 5 GB RAM, and runs the 64-bit Ubuntu Linux operating system.

These performance evaluation results are obtained for the following three different configurations:

1 the test application is not modified (labelled ‘unmodified application’ in the figure)

2 the test application is modified to contain HMAC message signing (labelled ‘with HMAC’ in the figure)

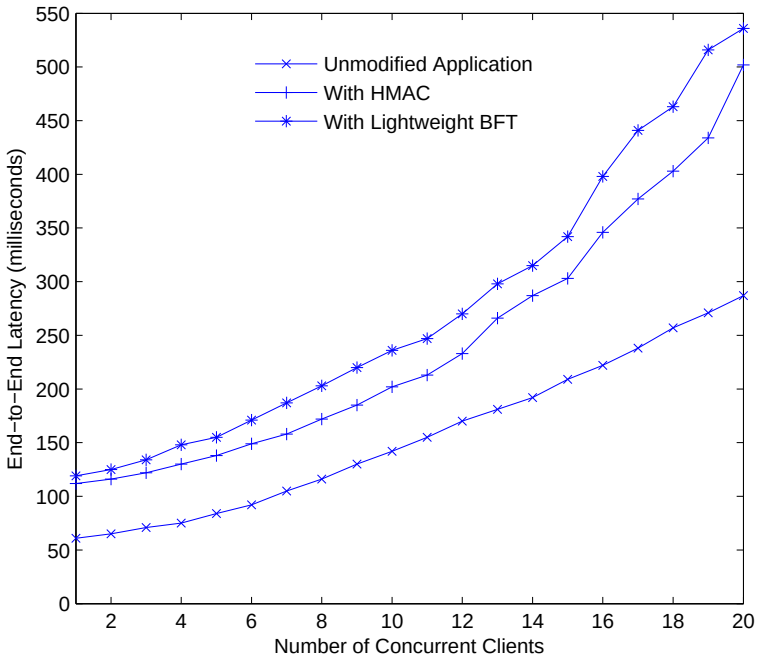
3 the test application is protected with our lightweight BFT framework (labelled ‘with lightweight BFT’ in the figure).

The end-to-end latency of each operation is measured at the client side. The throughput of the middle tier is measured at one of the shopping cart web service replicas. In each run, we obtained around 1,000 samples and calculated the median latency and the

median throughput. In all experiments, we assume that at most one replica can be faulty (i.e., $f = 1$).

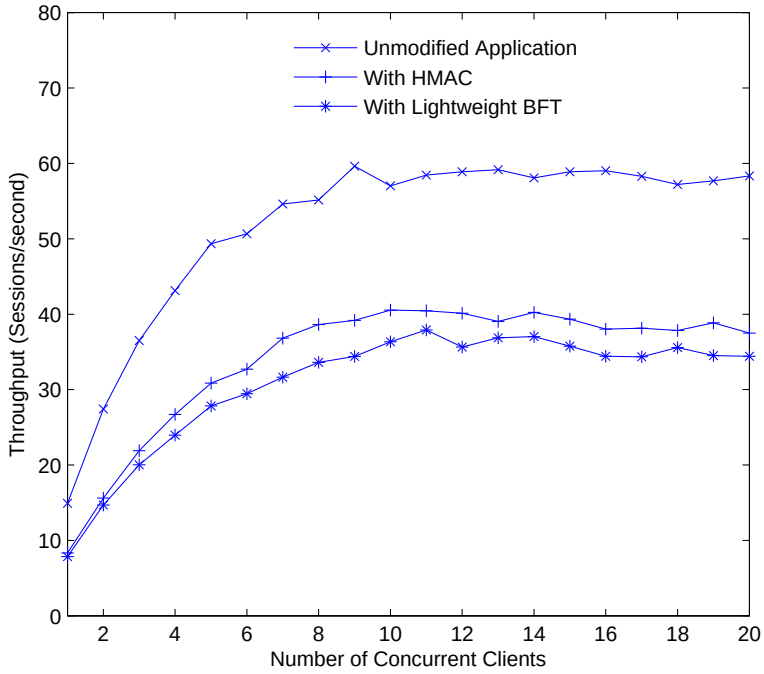
The end-to-end latency results are shown in Figure 4. As can be seen in Figure 4, the end-to-end latency for each session (which consists of 20 steps, as shown in Figure 2) with our lightweight BFT framework is significantly larger than that of the unmodified application. However, this increase is due mainly to the use of HMACs for message protection, as revealed by the similarly high end-to-end latency when only HMACs are used. The additional overhead incurred by our BFT framework (which is less than 15% over the second configuration) is primarily due to the voting steps taking place at the client and the backend server. The cost of securing messages varies by the message size. The cost of signing the 'browse' reply message in step 6 of Figure 2 is about 6 ms and verifying this message takes about 12 ms (a list of 50 item objects are contained in the 'browse' reply). The cost of securing other messages is similar. The signing takes about 1 to 2 ms and the verifying of each message takes about 1 ms. The cost of securing all the messages inevitably increases the end-to-end latency. However, it is essential to use HMAC (or RSA digital signature) for secure communication in practice. Thus, we use this configuration as the baseline for comparison.

Figure 4 End-to-end latency measured at a client with different number of concurrent clients (see online version for colours)



The throughput results (in terms of the number of sessions processed per minute) are shown in Figure 5. As can be seen in Figure 5, compared to the baseline configuration (with HMAC), the average throughput reduction with our lightweight BFT algorithm is less than 15%.

Figure 5 Throughput of the middle-tier server with different number of concurrent clients (see online version for colours)



5 Related work

There are a large body of work on modern BFT algorithms, such as (Castro and Liskov, 2002; Clement et al., 2009a, 2009b; Singh et al., 2009). These algorithms are designed to protect generic stateful servers against Byzantine faults in a client-server environment. It is not our goal to develop a competing BFT algorithm for generic stateful servers. Instead, this work is a continuation of our line of work on developing more efficient BFT solutions by exploiting application semantics.

In our previous work (Chai et al., to appear; Zhang et al., 2012), we argued that there is no need to apply expensive traditional BFT algorithms to web services business activity (WSBA) and web services atomic transaction (WSAT) applications, and we described a set of BFT mechanisms that can be used to enhance the trustworthiness of such applications. In this article, we focus on developing a lightweight BFT solution for building trustworthy session-oriented multi-tiered applications. The target applications of this paper involve a single user for each session, whereas multiple participants are involved in the WSBA and WSAT applications. Hence, in a way the target applications of this paper are simpler than the WSBA and WSAT applications.

We have also seen other related work that aimed to improve the performance of BFT systems by exploiting application semantics. In Kotlan and Dahlin (2004), the semantics of the networked file system is utilised to parallelise the execution of non-conflicting requests. However, to ensure correct partial ordering of conflicting requests, all requests have to be totally ordered in the first place, which may incur additional latency. In Distler

and Kapitza (2011), a further improvement is proposed by executing at a subset of the replicas, for networked file systems or similar applications. Again, all requests must be totally ordered in the first place. In sharp contrast, we observed that for (single-client) session-oriented multi-tiered applications, it is unnecessary to totally order the requests. As shown in our performance evaluation study, the runtime overhead of our approach is significantly less than that of the total-ordering approach. Furthermore, avoiding total ordering of requests also eliminates the uncertainty of potentially extensive unavailability of the BFT system due to view changes, which can be a serious concerns in the approaches that rely on the total ordering of requests.

6 Conclusions and future work

In this article, we have presented a lightweight BFT solution for session-oriented multi-tiered applications. The insight was obtained by carefully analysing the state model of such applications and potential threats to the system. We argue that it is unnecessary to perform total ordering of all incoming messages to the middle-tier server replicas, to achieve BFT. Rather, it suffices to ensure that the messages are delivered in source order, i.e., the order in which the sender sends them. Performance evaluation of the research prototype using a shopping cart application has shown that our lightweight BFT algorithm incurs very moderate (less than 15%) runtime overhead.

We plan to conduct future work in two fronts:

1 study other types of distributed systems, such as BPEL (Alves et al., 2007) applications, and identify new mechanisms based on the application semantics that be used to significantly lower the end-to-end latency and increase the system throughput

2 explore ways to automate or semi-automate the process of extracting ordering rules and execution rules based on the source code, and/or high level specifications (such as WSDL documents) of the applications, which could reduce the development cost of lightweight BFT solutions.

Acknowledgements

This work was supported in part by NSF grant CNS 08-21319, and by a CSUSI grant from Cleveland State University.

References

Alves, A., Arkin, A., Askary, S., Barreto, C., Bloch, B., Curbera, F., Ford, M., Goland, Y., Guizar, A., Kartha, N., Liu, C., Khalaf, R., Konig, D., Martin, M., Mehta, V., Thatte, S., Rijin, D., Yendluri, P. and Yiu, A. (Eds.) (2007) *Web Services Business Process Execution Language*, Version 2.0, OASIS Standard.

Apache Axis2 [online] http://ws.apache.org/axis2/ (accessed January 2012).

Apache Derby [online] http://db.apache.org/derby/.

Apache WSS4J [online] http://ws.apache.org/wss4j/ (accessed January 2012).

Castro, M. and Liskov, B. (2002) 'Practical Byzantine fault tolerance and proactive recovery', *ACM Transactions on Computer Systems*, Vol. 20, No. 4, pp.398–461.

Chai, H., Zhang, H., Zhao, W., Melliar-Smith, P.M. and Moser, L.E. (to appear) 'Toward trustworthy coordination for web service business activities', *IEEE Transactions on Services Computing*.

Clement, A., Kapritsos, M., Lee, S., Wang, Y., Alvisi, L., Dahlin, M. and Riche, T. (2009a) 'UpRight cluster services', *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, Big Sky, MT.

Clement, A., Wong, E., Alvisi, L. and Dahlin, M. (2009b) 'Making Byzantine fault-tolerant systems tolerate Byzantine faults', *Proceedings of the 6th Symposium on Networked Systems Design and Implementation*, Boston, MA.

Davis, D., Karmarkar, A., Pilz, G., Winkler, S. and Yalcinalp, U. (2008) *Web Services Reliable Messaging*, Version 1.1, OASIS Standard.'

Distler, T. and Kapitza, R. (2011) 'Increasing performance in Byzantine fault-tolerant systems with on-demand replica consistency', *Proceedings of the 6th Eurosys Conference*.

Kotlan, R. and Dahlin, M. (2004) 'High throughput Byzantine fault tolerance', *Proceedings of the International Conference on Dependable Systems and Networks*.

Lamport, L., Shostak, R. and Pease, M. (1982) 'The Byzantine generals problem', *ACM Transactions on Programming Languages and Systems*, Vol. 4, No. 3, pp.382–401.

Nadalin, A., Kaler, C., Monzillo, R. and Hallam-Baker, P. (2006) *Web Services Security: SOAP Message Security 1.1*, OASIS Standard.

Singh, A., Fonseca, P., Kuznetsov, P., Rodrigues, R. and Maniatis, P. (2009) 'Zeno: eventually consistent Byzantine-fault tolerance', *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*.

Zhang, H., Chai, H., Zhao, W., Melliar-Smith, P.M. and Moser, L.E. (2012) 'Trustworthy coordination for web service atomic transactions', *IEEE Transactions on Parallel and Distributed Systems*, Vol. 23, No. 8, pp.1551–1565.