2012 31st International Symposium on Reliable Distributed Systems

GRADE: Graceful Degradation in Byzantine Quorum Systems

Jingqiang Lin[1], Bo Luo[2], Jiwu Jing[1] and Xiaokun Zhang[3]
1. State Key Laboratory of Information Security, Institute of Information Engineering,
Chinese Academy of Sciences, Beijing 100195, CHINA
2. Department of Electrical Engineering and Computer Science, the University of Kansas, KS 66045, USA
3. Academy of Opto-Electronics, Chinese Academy of Sciences, Beijing 100094, CHINA
Email: linjq@lois.cn, bluo@ku.edu, jing@lois.cn, xkzhang@aoe.ac.cn

***Abstract*—Distributed storage systems are expected to provide correct services in the presence of Byzantine failures, which do not have any assumptions about the behavior of faulty servers and clients. In designing such systems, we often encounter the paradox of fault tolerance *vs.* performance (or efficiency), because better fault tolerance usually requires a tradeoff of system performance. In this paper, we present GRADE, a Byzantine-fault-tolerant (BFT) distributed storage system that enables graceful degradation. Two Byzantine quorum systems (BQSs) are supported on each GRADE server: a masking BQS storing generic data and a dissemination BQS storing self-verifying ones. Based on the system status and the environment, servers dynamically and seamlessly switch between two BQSs, without converting the stored data. Therefore, GRADE provides high performance in a normal running-state, and degrades performance to maintain high fault tolerance in emergency situations. The computation and communication costs of the running-state switch are very low, and the switch is completely transparent to clients. Our performance analysis and experimental results demonstrate that GRADE provides a balance between performance and fault tolerance.**

***Index Terms*—Byzantine fault tolerance; Byzantine quorum system; graceful degradation; storage.**

I. Introduction

Quorum systems are widely used to ensure availability and consistency of replicated data in distributed services. Among such systems, Byzantine quorum systems (BQSs) are capable of tolerating arbitrary failures (i.e. Byzantine failures) [1]. Several variations of BQSs with different capacities of fault tolerance and performance have been proposed [1–6]. These variations of BQSs are used to build Byzantine-fault-tolerant (BFT) distributed storage services [7–10], which do not have any assumptions on the behavior of faulty servers and clients.

When we select a variation of BQSs for BFT storage services, the capacity of fault tolerance is an important reference. In particular, the selection usually depends on the number of faulty servers in the worst scenario. For instance, if there might be at most $\lfloor \frac{n-1}{3} \rfloor$ faulty servers in a system consisting of n servers, we need a BQS tolerating up to $\lfloor \frac{n-1}{3} \rfloor$ faulty servers, even though the system contains no faulty servers in most of its running time.

Performance (or efficiency) is another important metric of storage systems. The BQS variants produce very different performance [11], and performance is usually sacrificed in exchange for better fault tolerance [12]. As a result, system performance has to be sacrificed to satisfy the fault tolerance requirements in rare situations (i.e., the worst case).

Based on the above observations, we propose GRADE, a BFT distributed storage system that enables graceful degradation [13]. GRADE consists of n servers capable of running in two different states, called *d-State* (resembling dissemination BQSs storing *self-verifying* data) and *m-State* (resembling masking BQSs storing *generic* data), respectively. Different types of data are stored in these running-states, which provide different capacities of fault tolerance as well as performance.

GRADE implements graceful degradation by *seamlessly* switching the running-state, without interrupting its storage services: (*a*) when the system starts up or is recovered by a periodical proactive recovery [7, 14, 15], GRADE runs in *m-State* to offer *better performance*; and (*b*) when GRADE is aware of a trigger event, it switches to *d-State* for *better fault tolerance* but relatively *lower performance*. In GRADE, a trigger event indicates a security condition that could result in more faulty servers than the capability of m-State, so that it is necessary for the system to trade off performance for better fault tolerance. Examples of such trigger events include: (*a*) a vulnerability is found in some servers but the patch is not available yet; (*b*) worms break out and will probably block the communications of servers; and (*c*) servers have run for a scheduled period since the last proactive recovery. In other words, when a trigger event takes place, the number of faulty servers is likely to increase soon and then break the assumption of m-State.

On a trigger event, GRADE servers follow the graceful degradation specification (i.e. the *running-state switch* protocol) to switch to d-States. During the switch, a naive solution will convert all the stored data from generic data (for masking BQSs or m-State) to self-verifying ones (for dissemination BQSs or d-State). However, this simple approach is unacceptable in practice, because the data conversion is very expensive, especially for the massive data stored in the system.

In order to tackle the problem of data type, we propose a lazy-conversion approach: (*a*) no data conversion is needed in graceful degradation, so the switch protocol is very efficient and scalable; and (*b*) availability and consistency are still ensured in d-State, in the presence of up to $\lfloor \frac{n-1}{3} \rfloor$ faulty

1060-9857/12 $26.00 © 2012 IEEE
DOI 10.1109/SRDS.2012.34

servers. These benefits come at a very low price: GRADE tolerates up to $\lfloor \frac{n-1}{6} \rfloor$ faulty servers in m-State, which is less than $\lfloor \frac{n-1}{4} \rfloor$ faulty servers in a regular masking BQS. Besides, the protocol for dissemination BQSs [7] is slightly revised as the storage protocol of GRADE in d-State to handle the unconverted generic data, which are written in m-State (see Section III for details).

Through graceful degradation, GRADE offers better performance than storage systems based on dissemination BQSs only, while providing almost equal capabilities of Byzantine fault tolerance. In particular, up to $\lfloor \frac{n-1}{3} \rfloor$ faulty servers are tolerated in d-State of GRADE as well as dissemination BQSs, but GRADE runs as a masking BQS when there is no risk (or trigger event); therefore, it offers better performance on average. On the other hand, compared with a masking BQS tolerating up to $\lfloor \frac{n-1}{4} \rfloor$ faulty servers, GRADE tolerates more faulty servers (i.e., up to $\lfloor \frac{n-1}{3} \rfloor$ in d-State) and then has a *longer* period of proactive recovery (PPR), because the PPR mainly depends on the number of faulty servers that the system tolerates [14, 16]. Note that in a distributed storage system, recovery is a very heavy task [7, 15]: (*a*) each server reboots from trust read-only medias, to enter a non-code-error status; (*b*) all data on each server are updated with the right copies, to enter a non-data-error status; and (*c*) servers re-generate or re-negotiate all cryptographic keys for communications, to prevent attackers from impersonating any server.

GRADE provides BFT distributed storage services as a regular BQS does, and supports graceful degradation with the following properties:

- *Low cost*. There is no data conversion in the running-state switch, and the switch costs even fewer resources than one read (or write) operation.
- *Service continuity.* Each server is always ready to process messages related to read and write operations. The storage service is not interrupted by the running-state switch.
- *Client transparency.* Different running-states are transparent to clients; i.e., all clients use a uniform protocol to access data stored in GRADE whether it is in d-State or m-State, even when the running-state is being switched.
- *Byzantine fault tolerance.* GRADE provides BFT storage services. Meanwhile, faulty servers cannot conspire to switch the system into d-State when there is no risk, to unnecessarily degrade its performance.

The rest of this paper is organized as follows. In Section II, the system model is presented. Section III shows how to implement graceful degradation in GRADE. Faulty clients are discussed in Sections IV, and the prototype implementation is evaluated in V. Related work is discussed in Section VI and we conclude in Section VII.

II. System Overview

GRADE consists of $n = 3f_d + 1$ servers and an arbitrary number of clients distinct from servers. Servers are correct or faulty. A correct server follows its specification and a faulty one can arbitrarily deviate from its specification (i.e., Byzantine failure). GRADE tolerates up to f_d faulty servers.

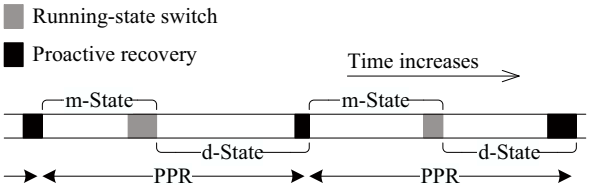

Fig. 1. The running-states of GRADE

We assume that a client always behaves correctly in Sections II and III, and faulty clients are discussed in Section IV.

GRADE servers are capable of dynamically switching between different BQSs: masking BQS and dissemination BQS. They are called *m-State* and *d-State* in GRADE, to reflect the fact that they are different from regular masking BQSs and dissemination BQSs.

- *m-State*. Similar to a *masking* BQS [1], GRADE stores *generic data* in m-State, and tolerates up to $f_m = \lfloor \frac{f_d}{2} \rfloor = \lfloor \frac{n-1}{6} \rfloor$ faulty servers. Please note that, compared with a regular masking BQS tolerating up to $\lfloor \frac{n-1}{4} \rfloor$ faulty servers, GRADE also needs to ensure data consistency with the running-state switch.
- *d-State*. Similar to a *dissemination* BQS [1], GRADE stores new data as *self-verifying* data, but the read operations support both self-verifying and generic data. In d-State, GRADE tolerates up to $f_d = \lfloor \frac{n-1}{3} \rfloor$ faulty servers as a dissemination BQS.

Each server independently maintains a local *state register*, which is changed as GRADE switches the running-state. The local register instructs a server to follow the storage protocol of d-State or m-State in each (read or write) operation. In GRADE, once a threshold number of servers change their state registers to d-State (and no data conversion is needed), the running-state switch is completed and the system is in d-State; otherwise, it is in m-State.

The system model of GRADE is similar to that of COCA [7] supporting proactive recovery in BQSs. Graceful degradation in GRADE works compatibly with proactive recovery: the running-state switches from m-State to d-State on a trigger event and recovers to m-State after each proactive recovery. When GRADE starts up or is proactively recovered, the state registers on all servers are set to m-State and it is in m-State. One instance of the switch protocol is executed during each PPR, to change the running-state from m-State to d-State (i.e., set the state registers to d-State). Thus, a state register is set to (*a*) d-State only in the running-state switch, or (*b*) m-State only when the server is periodically recovered (or initially starts up). In other cases, the state registers are kept read-only.

There are up to f_d faulty servers when GRADE is in d-State, and up to $f_m = \lfloor \frac{f_d}{2} \rfloor$ in m-State. As shown in Figure 1, the running-state switch is executed in m-State, and GRADE begins to be in d-State as long as the switch is completed. Note that there are up to f_m faulty servers during the running-state switch in m-State, and a faulty servers in m-State is still faulty in d-State within one PPR.

As those in regular BQSs, the data stored in GRADE can be viewed as variables supporting read and write operations. For a variable x, each server S_i stores a local copy denoted as $[x, v_i, t_i]$, where v_i is the value and t_i is the timestamp of x. In order to tolerate faulty servers, each read (or write) operation is executed on some read (or write) quorum of servers. For example, in a dissemination BQS, a new pair of value and timestamp $[x, v, t]$ must be delivered to a quorum of q_{dw} servers to complete the write operation, and the result of each read operation is chosen out of the copies from a quorum of q_{dr} servers. Similarly, we use q_{mw} and q_{mr} to denote the sizes of a write quorum and a read quorum in masking BQSs, respectively.

Accordingly, the server protocol of GRADE is composed of (*a*) the *storage protocol* to process requests from clients and maintain the local copies of variables, and (*b*) the *switch protocol* to manage the state registers. Further, the storage protocol is composed of two parts: the d-State part and the m-State one. A server decides to follow which part of the storage protocol, according to its own state register. The server protocol is described in Section III.

In order to achieve client transparency (and support proactive recovery [7, 8]), threshold signature schemes (TSSs) [17] are integrated with BQSs in GRADE. GRADE has a system-wide key pair, the *service private key* and the *service public key*. The service public key is known to all entities. The service private key is distributed among the n servers based on a TSS. Each server holds a partition of the service private key (called a *service key share*), and then any h ($1 < h \leq n$) servers can cooperatively use the service private key to sign messages while any subset of fewer than h servers cannot.

The service key pair is used to communicate with clients, and clients only accept messages verifiable with the service public key (i.e., signed using the service private key). In GRADE, the threshold to sign messages is $h = f_d + 1$, and then a signed message (e.g., a response to clients) implies that at least one correct server agrees with the content and ensures its correctness. The service private key is also used to sign data (to make them self-verifying) and tokens in the running-state switch; see Section III for more details.

The communications among servers are authenticated; i.e., a correct server receives a message from another server only if the other server sends it. The authenticated communications are implemented through public key cryptography [7, 9] or symmetric cryptography [15, 18]; e.g., a server signs messages using its private key (different from the service private key), and others can authenticate the origin. In the remainder, unless it is explicitly noted, a "signed" message is signed using the service private key (but not any server's private key).

Every server can authenticate messages from clients. For example, every client can hold a key pair to sign requests, and servers authenticate them by verifying the signatures. However, only the service public key is configured on clients. Clients don't (need to) know any server's public key or share session keys with any server; otherwise, these keys shall be refreshed and re-distributed to clients in each proactive recovery [7, 15].

Only asynchronous fair links [7, 8] are assumed to be provided among all entities. A fair link is a channel that may not deliver all messages sent, but if an entity keeps sending a message to another entity then it will be correctly delivered to the receiver eventually. In addition, the link is asynchronous; i.e., there is no bound on message delivery delay or server execution speed.

III. Graceful Degradation in GRADE

We firstly describe the BFT storage services of GRADE, and explain how to implement graceful degradation (i.e., switch a BQS to another one; these two BQSs store different types of data) without interrupting the storage services.

A. Storage Service

GRADE provides BFT storage services similar to those of Phalanx [9], COCA [7] and BFT-BC [19]. The COCA protocol designed for dissemination BQSs, is (*a*) adopted in GRADE as the d-State part of the storage protocol and (*b*) extended for generic data as the m-State part.

Client Protocol

To read a variable x, a client periodically sends a request $Req(R)$ to $f_d + 1$ servers, until it receives a signed read response $Resp(R)$ returning the copy written by the most recently completed operation (called the *right copy*).

To write a variable x, a client firstly reads it to return a response $Resp(R)$ and the current right copy is $[x, v, t']$. Then, it periodically sends a write request $Req(W)$ to f_d+1 servers, until receiving a signed write response $Resp(W)$. $Resp(R)$ is included in the write request $Req(W)$, and the new timestamp t_w of the variable is derived from $Req(W)$ and (the previous timestamp t' in) $Resp(R)$. In particular, a timestamp t is an ascending sequence number appended with an identifier; i.e., $t = t.seq|t.u$. And $t_w = (t'.seq + 1)|Hash(Req(W))$, where $Hash(\cdot)$ is a collision-free hash function.

Storage Protocol

GRADE guarantees that a read response returns the right copy that (*a*) is written by some client and (*b*) has the highest timestamp in the system, by executing each read (or write) operation on some quorum of servers.

Since each client request is sent to $f_d + 1$ servers, at least one correct server receives the request. The server is called a *delegate* for this read (or write) operation. Because clients only know the service public key (but not any other keys of servers), the delegate coordinates the requested operation among servers on behalf of the client sending the request. Note that the delegate is not a special or additional server; otherwise, it would be a vulnerable component not tolerating failures. On receiving a client request, each (correct) server becomes a delegate for it.

In GRADE, a delegate initiates the storage protocol among servers to construct and sign a response as follows:

1. The delegate executes the requested read (or write) operation on some quorum of servers. That is, it forwards the

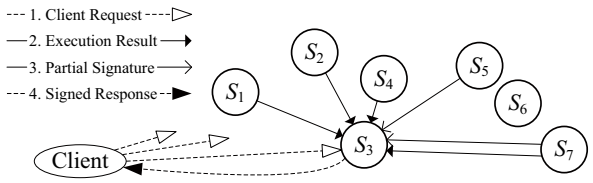

Fig. 2. The storage services of GRADE

client request to all servers, and collects the execution results from a quorum of servers.

2. The delegate constructs a raw response based on the execution results, and cooperates with other $h-1$ servers to sign it. That is, it sends the raw response to all servers, and waits from h partial signatures to combine them into a signed response.

Then, the delegate sends the signed response to clients. The procedure is shown in Figure 2. There are some notes on the storage protocol as follows, and the detailed security analysis can be found in [1, 7, 20]:

- The storage protocol is parameterized by the running-state of GRADE, and each server follows the protocol according to its own state register. In particular, every message among servers carries a flag indicating the sender server's state register, and a server doesn't accept messages from any server with a state register different from its own. We temporarily assume that the state registers on all servers are identical, and the case of different state registers is discussed later.
- The size of a read (or write) quorum of in m-State is q_{mr} (or q_{mw}), while the size in d-State is q_{dr} (or q_{dw}). In different BQSs, the result functions [1] to choose the right copy out of copies from servers are different: (*a*) in m-State or masking BQSs, the delegate discards those copies only returned by less than f_m+1 servers, and the right copy is the one with the highest timestamp of the remainders; and (*b*) in d-State or dissemination BQSs, the right copy is the self-verifying one (i.e., verifiable with the service public key) with the highest timestamp.
- In Step 2, the execution results collected in Step 1, are sent by the delegate along with the raw response as evidences to convince other servers to sign it. Before using its service key share to partially sign the raw response, a server firstly examines that the corresponding read (or write) operation has been executed on a quorum; i.e., (*a*) the execution results from some quorum of servers are included as evidences, and (*b*) in the case of read operations, the raw response includes the right copy chosen out of these execution results. Because there are up to f_d faulty servers and the threshold to sign is f_d+1, at least one correct server carries out the examination and the requested operation is guaranteed to be executed on some quorum of servers when the response is signed.
- GRADE stores self-verifying data in d-State. So, when the delegate's state register is d-State (and the state registers on other servers are also d-State), the delegate firstly cooperates with other servers to use the service private key to sign the new pair of value and timestamp into a self-verifying copy, before writing it to servers.
- When a delegate sends a message to servers, the message is also "sent" to itself and the delegate processes the message as other servers. In particular, the delegate also executes the read (or write) operation and partially signs the raw response.
- The timestamp and the value of each variable are initialized to zero. When a client reads a variable for the first time[1] and a server cannot find its local copy, the special initial copy is returned.

B. Switch Protocol

Graceful degradation is implemented by the running-state switch. The switch protocol is designed to change the state registers on servers. The basic idea is to use the service private key to sign a *switch token*, and each server sets its state register to d-State on receiving the signed switch token.

The sketch of the switch protocol is as follows:

1. If a server notices a trigger event, it (called the *initiator* of the switch protocol) sends a token-signing request to all servers (including itself). In the token-signing request, a credential is included, e.g., the URLs of web pages describing the vulnerabilities or the worms that might compromise servers.
2. On receiving a token-signing request, a server generates a partial signature for the switch token and sends it to the initiator, after validating the included credential.
3. The initiator periodically sends the token-signing request until receiving h partial signatures. Then, it combines them into a signed switch token.
4. The initiator periodically sends the signed token to all servers (also including itself), until receiving echoes from $n - f_m$ servers.
5. On receiving a signed switch token, a server sets its state register to d-State and replies with an echo.

In GRADE, multiple servers may simultaneously initiate the switch protocol; e.g., they notice a same trigger event. Whether its state register is m-State or d-State, a server always (*a*) partially signs the switch token after validating the credential, and (*b*) sets its state register to d-State and replies with an echo after receiving a signed switch token. Thus, all instances of the switch protocol will end with enough echoes.

C. Security Analysis

In the presence of up to f_m Byzantine faulty servers, the switch protocol shall satisfy the following requirements:

- *Byzantine fault tolerance*. Faulty servers cannot conspire to change the state registers of correct servers.
- *Liveness*. A correct server eventually receives $n - f_m$ echoes, once it initiates the switch protocol.

[1]To write a variable, the client shall read it firstly.

- *Completeness*. After the switch protocol is finished and before the next proactive recovery, all read and write requests from clients are processed in d-State to tolerate up to f_d faulty servers.
- *Compatibility*. The running-state switch doesn't impact liveness and correctness of the storage services. That is, either before, during or after the running-state switch, each read (or write) request gets a signed response, and each read response returns the right copy.

Firstly, Byzantine fault tolerance and liveness are satisfied. To finish the switch protocol, $f_d + 1$ and $n - f_m$ servers are required to sign the switch token and send echoes, respectively. Because there are up to f_d faulty servers in GRADE, faulty servers cannot conspire to sign switch tokens. Because there are up to f_m faulty servers during the running-state switch and $n - f_m = 3f_d + 1 - f_m > f_d + 1$, there are always enough correct servers to generate partial signatures and echoes.

To satisfy completeness, a client request is never completely processed in m-State after the switch protocol is finished. After the switch protocol is finished, $n - f_m$ servers have received the switch token, of which there are f_d faulty servers at most. Thus, completeness requires that

$$q_{mr} + (n - f_m) - n > f_d \tag{1}$$

$$q_{mw} + (n - f_m) - n > f_d \tag{2}$$

and then any quorum of q_{mr} (or q_{mw}) servers contains at least one *correct* that had received the switch token. This correct server doesn't accept messages with m-State flags, and then no read or write operation in m-State can be completed.

Compatibility implies correctness of the storage protocol. To tolerate up to f_d faulty servers in d-State, each quorum consists of $2f_d + 1$ servers out of $n = 3f_d + 1$ ones [1]:

$$q_{dr} = q_{dw} = 2f_d + 1 \tag{3}$$

To tolerate up to f_m faulty servers in m-State, any read quorum intersects a write quorum in at least $2f_m + 1$ servers (i.e., at least $f_m + 1$ correct ones) [1]:

$$q_{mr} \leq n - f_m, \text{ and } q_{mw} \leq n - f_m \tag{4}$$

$$q_{mr} + q_{mw} - n \geq 2f_m + 1 \tag{5}$$

However, Equations 3, 4 and 5 are not enough to ensure compatibility. When the running-state has been switched to d-State but no write operation in d-State is executed, all data are still not self-verifying[2] and then the result function of dissemination BQSs becomes invalid. In this case, a delegate of read operations applies a revised result function: the right copy is the one returned by $f_d + (n - q_{mw}) + 1$ servers. In the worst case, any other copy is returned by at most $f_d + (n - q_{mw})$ servers: f_d faulty servers[3] and $n - q_{mw}$ correct servers not involved in the last write operation in m-State.

[2]It is impractical to sign each data item (to make it self-verifying) during the switch, because there can be lots of data stored in the system.

[3]Within one PPR, a faulty servers in m-State is still faulty in d-State.

TABLE I
THE PARAMETERS OF GRADE

Parameter	Description
$n = 3f_d + 1$	The number of servers in GRADE.
f_d	The number of faulty servers that GARDE tolerates in d-State.
$q_{dr} = 2f_d + 1$ $q_{dw} = 2f_d + 1$	The size of a read (or write) quorum in d-State.
$f_m = \lfloor \frac{f_d}{2} \rfloor$	The number of faulty servers that GARDE tolerates in m-State.
$q_{mr} = f_d + f_m + 1$ $q_{mw} = n - f_m$	The size of a read (or write) quorum in m-State.

In order to guarantee that the delegate can always find a copy returned by $f_d + (n - q_{mw}) + 1$ servers, every write operation in m-State needs to be executed on $2f_d + (n - q_{mw}) + 1$ servers at least, of which f_d servers may be faulty. Then,

$$q_{mw} \geq 2f_d + (n - q_{mw}) + 1 \iff q_{mw} \geq \frac{n+1}{2} + f_d \tag{6}$$

From Equations 4 and 6, it is derived that

$$n - f_m \geq \frac{n+1}{2} + f_d \iff f_m \leq \frac{f_d}{2} \tag{7}$$

In order to achieve the greatest f_m (so that the system will stay in m-State as long as possible with better performance), $f_m = \lfloor \frac{f_d}{2} \rfloor$ and $q_{mw} = n - f_m$. Finally, the minimal q_{mr} to satisfy Equations 1, 4 and 5, is $q_{mr} = f_m + f_d + 1$. All parameters of GRADE are listed in Table I.

The result function of the storage protocol in d-State is revised consequently. After receiving a read request from clients, the delegate firstly collects copies from $q_{dr} = 2f_d + 1$ servers. Then,

C1. If there is no self-verifying copy, then no write operation in d-State is executed after the running-state switch. So, the right copy is the one returned by $f_d + f_m + 1$ servers (if such a copy exists; otherwise, the delegate keeps reading copies from servers until it is returned).

C2. If not-less-than $f_d + 1$ servers return (different) self-verifying copies, some write operation has been executed in d-State because at least one correct server returns its self-verifying copy. The right copy is the self-verifying one with the highest timestamp. Note that a faulty server can return a self-verifying but out-of-date copy (e.g., a copy written in d-State during the last PPR), even when no write operation is executed after the switch.

C3. Otherwise, the delegate keeps reading copies from more servers, until (*a*) $f_d + f_m + 1$ servers return an identical copy, i.e., the right copy; or (*b*) $f_d + 1$ servers return self-verifying copies and the right copy is the self-verifying one with the highest timestamp. Note that at least $f_d + 1$ correct servers store self-verifying copies after a write operation is completed in d-State.

Then, the delegate selects q_{dr} execution results that contain (*a*) the $f_d + f_m + 1$ identical copies and no self-verifying copy, or (*b*) at least $f_d + 1$ self-verifying copies. The right copy is contained in these q_{dr} execution results, and any other servers can apply the revised result function to choose it. These

execution result are sent to convince other servers to sign the read response.

When all servers are correct and the links are reliable, q_{dr} copies from servers are enough to choose the right copy. But if some servers are faulty or the links are unreliable, f_d more copies may be needed for the revised result function. Thus, compared with regular dissemination BQSs, performance of GRADE may be slightly degraded in case of read operations.

Because a server always sets its state register to d-State on receiving a signed switch token, the following design ensures liveness of the storage protocol during the running-state switch (i.e., when the state registers on servers are different).

- If the state register on a server is d-State and it receives messages with m-State flags, it replies with the signed switch token that it has received. If its state register is m-State and it receives messages with d-State flags, it asks the delegate to send the signed switch token firstly.
- If the state register on a delegate is d-State and some server asks for the switch token, it replies with the switch token. If its state register is m-State and it receives a signed switch token from other servers, it firstly sets its state register to d-State and re-starts the storage protocol.

D. Credential, Switch Token and Trigger Event

The switch protocol is initiated in the following cases during each PPR: (*a*) GRADE has run for a scheduled period of time after the last proactive recovery[4], or (*b*) another trigger event happens before the scheduled time. A special code is used as the credential of a scheduled switch, when a server initiates the switch protocol. To validate such credentials (or determine whether it is time to switch the running-state), all correct servers shall share a trust global time or there are only limited differences on their local clocks, which can be calibrated in proactive recovery. For other trigger events, the credentials are messages that can be validated by any server (e.g., the URLs of web pages describing the vulnerabilities).

The signed switch token enables a server to immediately set its state register to d-State, before the scheduled time. The signed switch token is useful, even if there is no other trigger event except the scheduled switch. For example, if the switch token is eliminated in GRADE and each server automatically change its state register at the scheduled time, a delegate in d-State can only wait when it finds that other servers are in m-State (due to the differences on their local clocks). On the contrary, the signed switch token in GRADE accelerates the running-state switch.

To prevent faulty servers from re-using a signed switch token in the next PPRs, an expiration time is embedded in each switch token (i.e., the time to start the next periodical proactive recovery). Then, faulty servers cannot re-use it to unnecessarily degrade performance in next PPRs, after the state registers of all servers are recovered to m-State.

To speed up the running-state switch, a server sends the switch token to $n - f_m - 2$ servers (not including itself and the initiator), when (*a*) it receives a signed switch token and (*b*) its state register is changed from m-State to d-State. Then, a server may receive a switch token when its state register has been d-State, and it only replies with an echo and doesn't send the switch token to other servers again.

[4]In a certain environment, this period depends on f_m, the number of faulty servers tolerated in m-State.

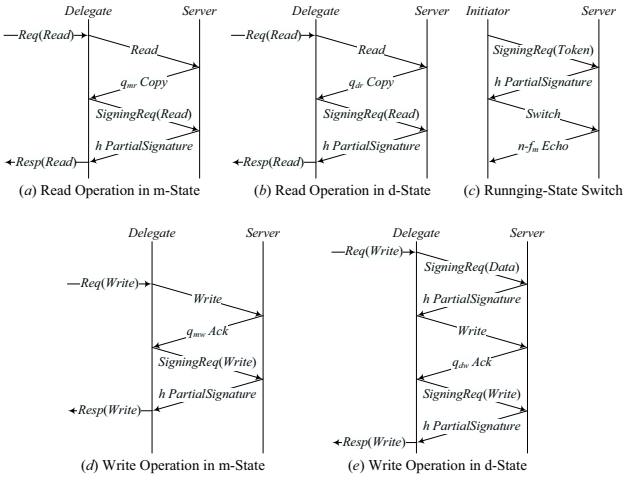

Fig. 3. The Communications of the Protocols

E. Performance Analysis

We analyze the performance of GRADE, and this analysis is confirmed by the experiments in Section V.

The detailed protocols are described in Appendix, and Figure 3 shows the sketch of the storage protocol and the switch protocol. Firstly, it can be found that (*a*) compared with that in m-State, each write operation in d-State needs one more round of communications to sign the self-verifying copy, and (*b*) each read operations in d-State needs $q_{dr} - q_{mr} \approx f_m$ more copies than that in m-State (i.e., the load[5] of servers is improved in m-State). So, GRADE has better performance when it is in m-State. Secondly, the switch protocol is composed of only two rounds of communications among servers, and the cost of communication and computation is comparable to that of read operations in GRADE.

IV. Faulty Client

As a storage system, GRADE cannot prevent a *malicious* client which strictly follows the client protocol, from writing erroneous values. However, GRADE needs to prevent *faulty*

[5]Given a quorum system, the load is the access probability of the busiest quorum, minimized over all strategies [21].

clients not strictly following the client protocol, from letting the system in an in-consistent state.

The following faulty-client-tolerant (FCT) mechanisms are designed in GRADE. Firstly, the mechanism of deriving a timestamp from the corresponding write request [7] guarantees that a faulty client cannot write different values with an identical timestamp to servers.

Secondly, faulty clients might (conspire with faulty servers to) execute a write operation on less than q_{mw} servers in m-State or less than q_{dw} in d-State. For example, a faulty client sends a write request to a faulty server, and the server writes it to less than q_{mw} servers. Then, faulty servers can conspire to manipulate the copy to be returned in the subsequent read operations. For example, when the copy with a higher timestamp is written to only f_m correct and f_m faulty servers in m-State, it is chosen as the right copy only if some faulty server sends the copy to the delegate; otherwise, another copy is chosen as the right one. The following mechanisms (similar to those in [1, 7, 19]) are employed to prevent such conspiracy attacks:

- On receiving a write request from delegates, a server in m-State updates its local copy and forwards it to $q_{mw}-2$ servers (not including the delegate and itself) if the copy to be written has a higher timestamp than its own. It guarantees that a write operation in m-State is executed on (*a*) faulty servers only or (*b*) at least q_{mw} servers if some correct server is involved.
- Similarly, on receiving a write request from delegates, a server in d-State updates its local copy and forwards it to $q_{dw}-2$ servers (not including the delegate and itself) if the self-verifying copy to be written has a higher timestamp. It guarantees that a write operation in d-State is executed on (*a*) faulty servers only or (*b*) at least q_{dw} servers.

Finally, a self-verifying copy in d-State could be chosen as the right copy even if it is returned by only one (faulty) server, so the write-back design [9, 19] is also needed: when partially signing a read response in d-State and the right copy is a self-verifying one, a server updates its local copy and writes it to $q_{dw}-2$ servers (not including the delegate and itself) if the right copy has a higher timestamp than its own. So, when a self-verifying copy is returned to clients, it has been written to some quorum of q_{dw} servers.

V. Implementation and Evaluation

In this section, we present the prototype implementation of GRADE and analyze the experiment results.

A. Prototype Implementation

The GRADE prototype consists of seven servers, tolerating two faulty servers in d-State and one faulty server in m-State. The 1024-bit RSA service key pair is shared based on Shoup's TSS [22]. Each server also holds a 1024-bit RSA key pair for authenticated communications among servers, and every client holds a 1024-bit RSA key pair to sign client requests. A variable has a 32-bit unique identity and a 512-bit value. Each timestamp is 192-bit, consisting of a 32-bit sequence number and a 160-bit SHA-1 hash value.

TABLE II
The Execution Time of Each Operation in GRADE

FCT	Operation	Running-State	Time (in ms)
Enabled	Read	m-State	146.80
		d-State	160.45
	Write	m-State	160.30
		d-State	298.00
Disabled	Read	m-State	146.25
		d-State	159.15
	Write	m-State	151.15
		d-State	286.95
-	Switch	-	94.70

Servers and clients communicate via UDP, using the protocols in Appendix. Besides, the following optimizations are implemented in the prototype, not harming its correctness and fault tolerance:

O1. A client firstly sends its request to only one server, to reduce redundant computations and communications. The request is sent to f_d+1 servers periodically, only if it doesn't receive the signed response within a scheduled period of time.

O2. For the read operations in m-State, a delegate can choose the right copy even when it receives only $q_{mr}-f_m = 2f_m+1$ copies. If a copy is returned by f_m+1 servers and has the highest timestamp, it is the right copy no matter what are received from another f_m servers[6]. These $2f_m+1$ copies can also be used to convince other servers to sign the read response.

B. Experiment

The prototype runs in a 100Mbps Ethernet. One client sends requests to servers and receives signed responses. The configuration of all servers and the client is Intel Pentium 4 (2.4GHz) CPU, 256MB RAM with Windows 2000 Server as the operation system.

All servers and the client are correct in these experiments. Table II shows the execution time of the storage protocol and the switch protocol, and each result is the average of 20 operations. The result of each read (or write) operation is measured on the client, starting when it sends the request and ending when it receives the signed response. For each read or write operation, the client sends only one request (i.e., the signed response is returned before it sends the request to more servers). In the read operations, the right copy in d-State is a self-verifying one. The execution time of the switch protocol is measured on the initiator, starting when it sends the token-signing request and ending when it receives enough echoes.

The experiment results confirm the performance analysis in Section III-E:

- Whether the FCT mechanisms are enabled or not, performance of write operations is much better in m-State. The

[6]The similar optimization cannot be applied to the read operations in d-State, because a self-verifying copy with the highest timestamp is the right copy even if it is returned by the last server of q_{dr} ones.

execution time in d-State is about 1.88 times as that in m-State, because one more round of communications among servers and one more threshold signing computation are needed.

- The performance improvement of read operations in m-State is not so remarkable as that of write operations, and the execution time in d-State is about 1.09 times as that in m-State. In d-State, more servers are involved in a read operation, and a self-verifying copy shall be verified by servers using the service public key.
- The running-state switch is completed much faster than a read (or write) operation. Compared with a read operations in m-State, the execution time is only 65%.

VI. Related Work

The relaxation lattice method [13] was proposed to specify graceful degradation (i.e., the degraded behaviors when the environment changes) for several applications. D-GRAID [23] implements graceful degradation in RAID arrays, to ensure availability of as many frequently-used files as possible when disks fail. A. Fox and E. Brewer [24] suggested to improve availability by degrading completeness of responses in large-scale services, in the presence of failures. BFT2F [25] provides degraded consistency when there are more than f but not more than $2f$ Byzantine faulty servers out of $3f + 1$ servers, and L. Zhou *et al.* extended the notion of linearizability to three degraded semantics in state machine replication [26]. This work investigates graceful degradation from a different view: performance is degraded to tolerate more faulty servers when the number of faulty servers is (suspected to be) greater than the initial assumption.

Dynamic BQSs [27–29] reconfigure the number of faulty servers that it tolerates (e.g., in a dissemination BQS consisting of $3f_d + 1$ servers, the number can be any value between 1 and f_d), to achieve dynamic fault tolerance and improve the load of servers. However, they support only one fixed variation of BQSs, and then the execution time of each read or write operation is limitedly improved. Theoretically, this dynamic design can work compatibly in GRADE, and then GRADE servers run as different dynamic BQSs in each running-state.

COCA [7] and CODEX [8] integrate TSSs [17, 30] to support proactive recovery in dissemination BQSs. As mentioned above, the running-state switch protocol of GRADE works with proactive recovery to guarantee that the number of faulty servers never breaks the assumption. BFT-BC [19] is another protocol of dissemination BQSs, not integrating TSSs. A write response of BFT-BC is appended with signatures from $2f_d+1$ servers that accept the write request, while a write response of COCA is signed by $f_d + 1$ servers based on TSSs, each of which partially signs it after checking that $2f_d + 1$ servers have accepted the write request. Similarly, each self-verifying copy in BFT-BC is appended with $2f_d+1$ servers' signatures, while that in COCA is signed by f_d+1 servers cooperatively. Both of these protocols tolerate an arbitrary number of faulty clients and f_d faulty servers out of $3f_d+1$ ones. We adopt the COCA protocol as the storage protocol of GRADE in d-State, because TSSs are also needed in GRADE to achieve client transparency.

HQ [18] supports two approaches to implement BFT state machine replication: HQ employs a lightweight quorum-based protocol to provide better performance when there is no contention, and PBFT [15] is used to resolve contention when it raises. GRADE shares the same spirit that the system switches to different protocols for better performance as the environment changes. However, two BQSs with different capacities of fault tolerance are supported in GRADE, while HQ applies one BQS tolerating the same number of faulty servers as PBFT (i.e., it supports two states with the same fault tolerance).

We extend the preliminary work [31] of GRADE in three ways at least: (*a*) the FCT mechanisms are discussed only in this version, (*b*) the security analysis and the performance evaluation are more comprehensive, and (*c*) the discussion about credentials, switch tokens and trigger events is not presented in [31].

VII. Conclusion and Future Work

We present GRADE, a BFT distributed storage service, which is capable of dynamically switching between two different BQSs. GRADE provides higher performance in a normal running-state, and degrades performance to maintain high fault tolerance in emergency situations. The running-state switch in GRADE is very efficient and scalable, because it does not require any data conversion. The storage service is not interrupted by the running-state switch. We present the detailed protocols, and prove its security. Experimental results show that GRADE provides a balance between performance and fault tolerance.

With the first GRADE system being successfully implemented, we plan to extend the framework in several directions. We plan to support more different BQSs to provide fine-grained performance and fault tolerance balances. We are also investigating a more automatical and intelligent running-state switch scheme, in which GRADE servers switch states based on their evaluation of the environment risk; especially, servers can switch back to m-States when the risk becomes low, instead of waiting for a periodical proactive recovery.

Acknowledgment

The authors would like to thank Prof. Peng Liu of the Pennsylvania State University for helpful discussions. Jingqiang Lin and Jiwu Jing were partially supported by National Natural Science Foundation of China grant 70890084/G021102, 61003273 and 61003274, and Strategy Pilot Project of Chinese Academy of Sciences sub-project XDA06010702. Bo Luo was partially supported by NSF award OIA-1028098.

References

[1] D. Malkhi and M. Reiter, "Byzantine quorum systems," *Distributed Computing*, vol. 11, no. 4, pp. 203–213, 1998.

[2] R. Bazzi, "Synchronous Byzantine quorum systems," *Distributed Computing*, vol. 13, no. 1, pp. 45–52, 2000.
[3] J.-P. Martin, L. Alvisi, and M. Dahlin, "Minimal Byzantine storage," in *16th International Conference on Distributed Computing (DISC)*, 2002, pp. 311–325.
[4] ——, "Small Byzantine quorum systems," in *32nd International Conference on Dependable Systems and Networks (DSN)*, 2002, pp. 374–383.
[5] D. Malkhi, M. Reiter *et al.*, "Probabilistic quorum systems," *Information and Computation*, vol. 170, no. 2, pp. 184–206, 2001.
[6] M. Merideth and M. Reiter, "Probabilistic opaque quorum systems," in *21st International Symposium on Distributed Computing (DISC)*, 2007, pp. 403–419.
[7] L. Zhou, F. Schneider, and R. Renesse, "COCA: A secure on-line certification authority," *ACM Transactions on Computer Systems*, vol. 20, no. 4, pp. 329–368, 2002.
[8] M. Marsh and F. Schneider, "CODEX: A robust and secure secret distribution system," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 34–47, 2004.
[9] D. Malkhi and M. Reiter, "Secure and scalable replication in Phalanx," in *17th IEEE Symposium on Reliable Distributed Systems (SRDS)*, 1998, pp. 51–60.
[10] M. Abd-El-Malek, G. Ganger *et al.*, "Fault-scalable Byzantine fault-tolerant services," in *20th ACM Symposium on Operating Systems Principles (SOSP)*, 2005, pp. 59–74.
[11] W. Dantas, A. Bessani *et al.*, "Evaluating Byzantine quorum systems," in *26th IEEE Symposium on Reliable and Distributed Systems (SRDS)*, 2007, pp. 253–264.
[12] O. Rutti, Z. Milosevic, and A. Schiper, "Generic construction of consensus algorithms for benign and Byzantine faults," in *40th International Conference on Dependable Systems and Networks (DSN)*, 2010, pp. 343–352.
[13] M. Herlihy and J. Wing, "Specifying graceful degradation," *IEEE Transactions on Parallel and Distributed Systems*, vol. 2, no. 1, pp. 93–104, 1991.
[14] R. Ostrovsky and M. Yung, "How to withstand mobile virus attacks," in *10th ACM Symposium on Principles of Distributed Computing (PODC)*, 1991, pp. 51–59.
[15] M. Castro and B. Liskov, "Practical Byzantine fault tolerance and proactive recovery," *ACM Transactions on Computer Systems*, vol. 20, no. 4, pp. 398–461, 2002.
[16] R. Canetti and A. Herzberg, "Maintaining security in the presence of transient faults," in *Advances in Cryptology – Crypto'94*, 1994, pp. 424–438.
[17] Y. Desmedt, "Society and group oriented cryptography: A new concept," in *Advances in Cryptology – Crypto'87*, 1987, pp. 120–127.
[18] J. Cowling, D. Myers *et al.*, "HQ replication: A hybrid quorum protocol for Byzantine fault tolerance," in *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006, pp. 177–190.
[19] B. Liskov and R. Rodrigues, "Tolerating Byzantine faulty clients in a quorum system," in *26th IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2006, pp. 105–114.
[20] J. Lin, P. Liu *et al.*, "Impossibility of finding any third family of server protocols integrating Byzantine quorum systems with threshold signature schemes," in *6th ICST Conference on Security and Privacy in Communication Networks (SecureComm)*, 2010, pp. 307–325.
[21] M. Naor and A. Wool, "The load, capacity, and availability of quorum systems," *SIAM Journal of Computing*, vol. 27, no. 2, pp. 423–447, 1998.
[22] V. Shoup, "Practical threshold signatures," in *Advances in Cryptology – EuroCrypt'2000*, 2000, pp. 207–220.
[23] M. Sivathanu, V. Prabhakaran *et al.*, "Improving storage system availability with D-GRAID," *ACM Transactions on Storage*, vol. 1, no. 2, pp. 133–170, 2005.
[24] A. Fox and E. Brewer, "Harvest, yield and scalable tolerant systems," in *7th Workshop on Hot Topics in Operating Systems (HotOS)*, 1999, pp. 174–178.
[25] J. Li and D. Mazieres, "Beyond one-third faulty replicas in Byzantine fault tolerant systems," in *4th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2007, pp. 131–144.
[26] L. Zhou, V. Prabhakaran *et al.*, "Graceful degradation via versions: Specifications and implementations," in *26th ACM Symposium on Principles of Distributed Computing (PODC)*, 2007, pp. 264–273.
[27] L. Alvisi, M. Dahlin *et al.*, "Dynamic Byzantine quorum systems," in *30th International Conference on Dependable Systems and Networks (DSN)*, 2000, pp. 283–292.
[28] J.-P. Martin and L. Alvisi, "A framework for dynamic Byzantine storage," in *34th International Conference on Dependable Systems and Networks (DSN)*, 2004, pp. 325–334.
[29] L. Kong, A. Subbiah *et al.*, "A reconfigurable Byzantine quorum approach for the Agile Store," in *22nd IEEE Symposium on Reliable Distributed Systems (SRDS)*, 2003, pp. 219–228.
[30] L. Zhou, F. Schneider, and R. Renesse, "APSS: Proactive secret sharing in asynchronous systems," *ACM Transactions on Information and System Security*, vol. 8, no. 3, pp. 259–286, 2005.
[31] W. Wang, J. Lin *et al.*, "Graceful degradation in TSS-BQS systems (accepted to appear)," *Chinese Journal of Computers*.

APPENDIX

In this appendix, the client protocol and the server protocol are described in detail. The following notations are used:

- $[m]_{SK}$: Message m signed cooperatively by servers using the service private key.
- $[m]_c$: Message m signed by client c.
- S_i, S_d, S_a: A server, the delegate of the storage protocol, and the initiator of the switch protocol.
- $[m]_i, [m]_d, [m]_a$: Message m signed by S_i, S_d and S_a.
- $PS_i(m)$: A partial signature for message m, generated by S_i using its service key share.

A. Client Protocol

Read

1. To read x, c sends $Req(R) = [\texttt{Read}, c, x, p]_c$ to $f_d{+}1$ servers, where p is an ascending sequence number maintained by c.
2. c periodically sends $Req(R)$ to the $f_d + 1$ servers, until it receives $Resp(R) = [\texttt{Copy}, c, x, v, t, p]_{SK}$, where v is the value and t is the timestamp.

Write

1. To write x, c firstly sends $Req(R)$ to obtain $Resp'(R) = [\texttt{Copy}, c, x, v', t', p']_{SK}$.
2. c sends $Req(W) = [\texttt{Write}, c, x, v, p, Resp'(R)]_c$ to $f_d + 1$ servers, where v is the new value to be written.
3. c periodically sends $Req(W)$ to the $f_d + 1$ servers, until it receives $Resp(W) = [\texttt{Ack}, c, x, t_w, p]_{SK}$, where $t_w = t'.seq|Hash(Req(W))$.

B. Storage Protocol

Read in m-State

1. S_d sends $[\texttt{MRead}, d, Req(R)]_d$ to all servers, when it receives $Req(R)$ from c.
2. S_i replies with $[\texttt{MCopy}, i, [x, v_i, t_i], Req(R)]_i$, when it receives a MRead message from S_d, where $[x, v_i, t_i]$ is the local copy of x on S_i.
3. S_d repeats Step-1 periodically, until it receives MCopy messages from q_{mr} servers.
4. S_d chooses the right copy $[x, v, t]$ out of the q_{mr} ones and sends $[\texttt{MSignRead}, d, \bar{R}esp_R, \Sigma_{MR}, Req(R)]_d$ to all servers, where $\bar{R}esp_R = [\texttt{Copy}, c, x, v, t, p]$ and Σ_{MR} is the collection of q_{mr} MCopy messages.
5. S_i replies with $[\texttt{MPSRead}, i, PS_i(\bar{R}esp_R), Req(R)]_i$, when it receives a MSignRead message from S_d and checks that (*a*) Σ_{MR} are MCopy messages from q_{mr} servers for $Req(R)$ and (*b*) $\bar{R}esp_R$ includes the right copy.
6. S_d repeats Step-4 periodically, until it receives partial signatures from $f_d + 1$ servers. S_d sends $[\texttt{Copy}, c, x, v, p]_{SK}$ to c, after it combines the partial signatures into a signed read response.

Write in m-State

1. S_d sends $[\texttt{MWrite}, d, Req(W)]_d$ to all servers, when it receives $Req(W)$ from c.
2. S_i replies with $[\texttt{MAck}, i, x, t_w, Req(W)]_i$, when it receives a MWrite message from S_d, where $t_w = t'.seq|Hash(Req(W))$. Then, S_i updates its local copy to $[x, v, t_w]$ only if $t_w > t_i$.
3. S_d repeats Step-1 periodically, until it receives MAck messages from q_{mw} servers.
4. S_d sends $[\texttt{MSignWrite}, d, \bar{R}esp_W, \Sigma_{MW}, Req(W)]_d$ to all servers, where $\bar{R}esp_W = [\texttt{Ack}, c, x, t_w, p]$ and Σ_{MW} is the collection of of q_{mw} MAck messages.
5. S_i replies with $[\texttt{MPSWrite}, i, PS_i(\bar{R}esp_W), Req(W)]_i$, when it receives a MSignWrite message from S_d and checks that Σ_{MW} are MAck messages from q_{mw} servers for $Req(W)$.
6. S_d repeats Step-4 periodically, until it receives partial signatures from $f_d + 1$ servers. S_d sends $[\texttt{Ack}, c, x, t_w, p]_{SK}$ to c, after it combines the partial signatures into a signed write response.

Read in d-State

1. S_d sends $[\texttt{DRead}, d, Req(R)]_d$ to all servers, when it receives $Req(R)$ from c.
2. S_i replies with $[\texttt{DCopy}, i, Copy_i(x), Req(R)]_i$, when it receives a DRead message from S_d, where $Copy_i(x)$ is the local copy of x on S_i and it can be $[x, v_i, t_i]_{SK}$ or $[x, v_i, t_i]$.
3. S_d repeats Step-1 periodically, until it receives DCopy messages from q_{dr} servers.
   - C1. If there is no self-verifying copy, the right copy is the one returned by $f_d + f_m + 1$ servers.
   - C1. If there are not-less-than $f_d + 1$ self-verifying copies, the right copy is the self-verifying one with the highest timestamp.
   - C3. Otherwise, S_d repeats Step-1 periodically, until C1 or C2 happens.
4. S_d chooses the right copy $[x, v, t]$ and sends $[\texttt{DSignRead}, d, \bar{R}esp_R, \Sigma_{DR}, Req(R)]_d$ to all servers, where Σ_{DR} is the collection of q_{dr} DCopy messages containing (*a*) the $f_d + f_m + 1$ identical generic copies and no self-verifying copy or (*b*) at least $f_d + 1$ self-verifying copies.
5. S_i replies with $[\texttt{DPSRead}, i, PS_i(\bar{R}esp_R), Req(R)]_i$, when it receives a DSignRead message from S_d and checks that (*a*) Σ_{DR} are DCopy messages from q_{dr} servers for $Req(R)$ and (*b*) $\bar{R}esp_R$ includes the right copy.
6. S_d repeats Step-4 periodically, until it receives partial signatures from $f_d{+}1$ servers. S_d sends $[\texttt{Copy}, c, x, v, t, p]_{SK}$ to c, after it combines the partial signatures into a signed response.

Write in d-State

1. S_d sends $[\texttt{DSignData}, d, Req(W)]_d$ to all servers, when it receives $Req(W)$ from c.
2. S_i replies with $[\texttt{DPSData}, i, PS_i(x, v, t_w), Req(W)]_i$, when it receives a DSignData message from S_d.
3. S_d repeats Step-1 periodically, until it receives partial signatures from $f_d + 1$ servers and combines the partial signatures into a self-verifying copy $[x, v, t_w]_{SK}$.
4. S_d sends $[\texttt{DWrite}, d, [x, v, t_w]_{SK}]_d$ to all servers.
5. S_i replies with $[\texttt{DAck}, i, x, t_w, Req(W)]_i$, when it receives a DWrite message from S_d. Then, S_i updates its local copy to $[x, v, t_w]_{SK}$ only if $t_w > t_i$.
6. S_d repeats Step-4 periodically, until it receives DAck messages from q_{dw} servers.
7. S_d sends $[\texttt{DSignWrite}, d, \bar{R}esp_W, \Sigma_{DW}, Req(W)]_d$ to all servers, where Σ_{DW} is the collection of of q_{dw} DAck messages.
8. S_i replies with $[\texttt{DPSWrite}, i, PS_i(\bar{R}esp_W), Req(W)]_i$, when it receives a DSignWrite message from S_d and checks that Σ_{DW} are DAck messages from q_{dw} servers for $Req(W)$.
9. S_d repeats Step-7 periodically, until it receives partial signatures from $f_d + 1$ servers. S_d sends $[\texttt{Ack}, c, x, t_w, p]_{SK}$ to c, after it combines the partial signatures into a signed response.

C. Switch Protocol

1. S_a sends $[\texttt{GSignToken}, a, [\texttt{SToken}, et, sid], cred]_a$ to all servers, where $cred$ is the credential authorizing the running-state switch, et is the expiration time of the switch token, and $sid = Hash(cred, et)$ is the identifier of this switch.
2. S_i replies with $[\texttt{GPSToken}, i, PS_i(\texttt{SToken}, et, sid), sid]_i$, when it receives a GSignToken message from S_d.
3. S_a repeats Step-1 periodically, until it receives partial signatures from $f_d + 1$ servers and combines the partial signatures into a signed switch token.
4. S_a sends $[\texttt{GSwitch}, a, [\texttt{SToken}, et, sid]_{SK}]_a$ to all servers.
5. S_i replies with $[\texttt{GEcho}, i, sid]_i$ and sets its state register to d-State, when it receives a GSwitch message from S_a.
6. S_a repeats Step-4 periodically, until it receives GEcho messages from $n - f_m$ servers.