Vol. 44 No. 3 IPSJ Journal Mar. 2003

Regular Paper

Nested Invocation Protocol on Object Replicas

Kenichi Hori,† Tomoya Enokido† and Makoto Takizawa†

Objects are replicated in order to increase reliability and availability of an object-based system. If a method t is invoked on multiple replicas and each instance of t on the replicas invokes another method u on an object y, the method u is performed multiple times on the object of y although u should be performed just once. Then, the object gets inconsistent. This is redundant invocation. In addition, if each instance of the method t issues a request u to its quorum, more number of the replicas are manipulated than the quorum number of the method u. This is quorum expansion. We discuss a protocol to invoke methods on multiple replicas in a nested manner where the redundant invocation and quorum expansion are resolved. We evaluate the protocol compared with the primary-secondary replication.

1. Introduction

Objects are replicated in order to increase the reliability and availability in object-based applications [10]. There are many discussions on how to replicate state-full database servers [2),4)~7),11),12)], the two-phase locking [4] and quorum-based [6),7)] protocols. The quorum concept for read and write is extended to abstract methods supported by objects [11].

In the object-based system, an object is an encapsulation of data and methods. Furthermore, methods are invoked in a nested manner. Suppose a method t on an object x invokes a method u on another object y. Let x_1 and x_2 be replicas of the object x. Let y_1 and y_2 be replicas of y. In a primary-secondary way, x_1 and y_1 are primary replicas and the others are secondary ones. A method is invoked on a primary replica. The method t on the primary replica x_1 and then the method u on y_1. A checkpoint is taken on the primary replica and is transferred to secondary replicas. If a primary replica is faulty, one of the secondary replicas takes over the primary replica. The new primary replica restarts at the checkpoint. This approach is simple but less available. In order to increase the availability, each method is performed on more than one replica. A method t is issued to the replicas x_1 and x_2. A set of replicas x_1 and x_2 is referred to as *quorum* of the method t. We assume that every method is deterministic. Then, the method t invokes the method u on replicas y_1 and y_2. Here, the method u is performed twice on each replica. If multiple instances of the method u are performed on some replicas, the replicas may get inconsistent. This is a *redundant invocation.* In addition, an instance of the method t on the replica x_1 issues a method u to replicas in its own quorum Q_1, and another instance of t on x_2 issues u to replicas in Q_2 where $|Q_1| = |Q_2| = N_u$ but $Q_1 \neq Q_2$. More number of replicas are manipulated for a method u than N_u, i.e., $|Q_1 \cup Q_2| \geq N_u$. If the method u furthermore invokes another method, the number of replicas manipulated are more increased. This is a *quorum expansion.* In order to increase the reliability and availability, a method issued has to be performed on multiple replicas. On the other hand, the replicas may get inconsistent by the redundant invocations and the overhead is increased by the quorum expansion. We discuss how to resolve the redundant invocation and quorum expansion in nested invocations of methods on multiple replicas.

In section 2, we overview replication technologies. In section 3, we discuss what problems to occur in nested invocation of methods on replicas. In sections 4 and 5, we discuss how to resolve the redundant invocation and the quorum expansion, respectively. In section 6, we evaluate the quorum-based protocol.

2. Replication of Object

There are various kinds of discussions on how to replicate a system. As Wiesmann [12] discusses, there are different ways to replicate processes and database servers. Processes are stateless while database servers are statefull. There are three ways to replicate processes, *active*, *passive*, and *hybrid* ones. In the active replication [9], every replica receives a same se-

† Department of Computers and Systems Engineering, Tokyo Denki University

quence of messages, same computation is performed on every replica, and same outputs are sent back. Here, the process is required to be deterministic. The process is operational as long as at least one replica is operational. In the passive replication [3], there is one primary replica, say p_1, and the other replicas are secondary. Messages are sent to only the primary replica p_1 and the computation is performed on only the primary replica p_1. No computation is performed on any secondary replica. At a checkpoint of the primary replica p_1, a state of p_1 is sent to all the secondary replicas. The hybrid replication [1] is same as the passive one except that messages are sent to not only the primary replica but also the secondary replicas.

Ways to replicate database servers are classified with respect to which replica a request is issued to, *eager* and *lazy*, and when other replicas are updated, *primary* and *everywhere*. Requests are performed on replicas as soon as requests are issued in the eager type. On the other hand, requests are not immediately performed in the lazy one. In the primary replication, requests are performanced only on a primary replica. In the everywhere replication, requests are performed on all the replicas.

3. Nested Invocation on Replicas

3.1 Types of Method

Methods are procedures for manipulating objects. That is, methods are more complex and abstract than simple methods *read* and *write* on a file object. For example, a method *increment* on an object *counter* is realized by a sequence of *read* and *write* methods. There are *dependent* and *independent* types of methods. Computation of a dependent method t depends on object state. A method *increment* is a dependent one. Independent methods are performed independently of object state. There are furthermore *update* and *non-update* types of methods according to whether or not object state is changed by performing methods. For example, *increment* is a dependent update method since a *counter* value is incremented. A method *display* is a dependent, non-update one on an object *counter*. A method *append* is an independent, update method since data to be added is independent of object state while the state is changed.

3.2 Primary-secondary Invocation

Objects are encapsulation of data and methods for manipulating the data. Objects are manipulated only by invoking methods supported by the objects. Here, suppose a transaction T invokes a method t on an object x. Data in the object x is manipulated only by performing the method t. The method t is realized by invocations of other methods, say a method u on an object y. Thus, methods are invoked on objects in a *nested* manner (**Fig. 1**).

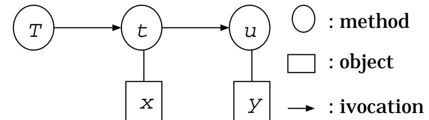

Fig. 1 Nested invocation.

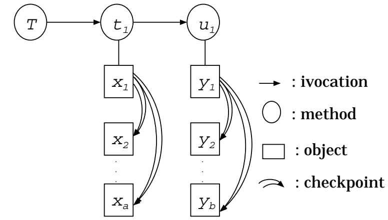

Fig. 2 Primary-secondary replication.

Suppose there are replicas $x_1, \ldots, x_a$ ($a>1$) of an object x and replicas $y_1, \ldots, y_b$ ($b>1$) of another object y. We discuss how to invoke methods on replicas of objects. In a *primary-secondary* one, the transaction T first issues a request t to only a primary replica x_1. Then, a request u in t is issued to a primary replica y_1 (**Fig. 2**). After the method commits, the state of the primary replica is eventually transmitted to the secondary ones. For example, a checkpoint [8] is taken on the primary replica and then the checkpoint data is transferred to secondary ones. Here, the secondary replicas catch up with the primary one. Since only one instance of t invokes u, neither redundant invocation nor quorum expansion occurs. For example, suppose a replica y_1 is faulty when t_1 invokes u_1 on y_1. One secondary replica, say y_2 is taken as the primary replica and t_1 invokes u_2 on y_2. The replica y_2 restarts on a previous state taken at the most recent checkpoint of y_1. Thus, the primary-secondary replication is less available due to the fault of primary replica.

3.3 Multi-invocation Model

We take another approach where a method is issued to multiple replicas in order to increase the reliability and availability (**Fig. 3**). Here, a transaction T invokes a method t on multiple replicas of an object x. Each instance t_i of t

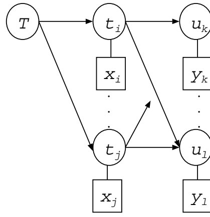

Fig. 3 Invocation on multiple replicas.

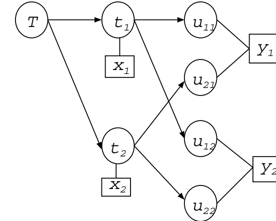

Fig. 4 Redundant invocation.

on a replica x_i invokes a method u on multiple replicas of another object y. Even if some replica is faulty, the method t is performed on other replicas and u is invoked on replicas of y. Let Q_{ui} be a *quorum* of a method u which is a subset of replicas of the object y to which an instance t_i issues a method u. Suppose there are four replicas y_1, y_2, y_3 and y_4. $Q_{u1} = \{y_1, y_2\}$ and $Q_{u2} = \{y_2, y_3\}$. That is, an instance t_1 invokes a method u on replicas y_1 and y_2, and another instance t_2 issues u on y_1 and y_2. Thus, an instance of the method u is performed on each replica in a subset $Q_{u1} \cup Q_{u2} = \{y_1, y_2, y_3\}$. $|Q_{u1} \cup Q_{u2}|$ $(= 3) \geq N_u$. This means that more number of replicas of y are manipulated than N_u. Then, the instances of u on the replicas in $Q_{u1} \cup Q_{u2}$ issue further requests to other replicas and more number of replicas are manipulated. The deeper level in a transaction, the more number of replicas are manipulated. This is *quorum expansion*. If Q_{ui} is not necessarily equal to a quorum Q_{uj} of another instance t_j, the quorum is expanded. $|Q_{ui} \cup Q_{uj}| > N_u$ may hold. Thus, the transaction T manipulates more number of replicas of the object y than N_u, i.e., the quorum of the method u is *expanded*.

Next, suppose a transaction T issues a method t to a pair of replicas in the quorum $Q_t = \{x_1, x_2\}$ and $N_t = 2$. Furthermore, the method t issues a request u to replicas of the object y in the quorum of u, say $N_u = 2$. Let t_i be an instance of the method t performed on a replica x_i ($i = 1, 2$). Each instance t_i issues a request u to replicas in a quorum Q_{ui}. Suppose $Q_{u1} = Q_{u2} = \{y_1, y_2\}$. Here, let u_{i1} and u_{i2} show instances of the method u performed on replicas y_1 and y_2, respectively, which are issued by a method instance t_i ($i = 1, 2$) (**Fig. 4**). Suppose the method u is "$y = 2{*}y$". However, the replica y_1 is multiplied by four since a pair of instances u_{11} and u_{21} are performed on y_1. Thus, y_1 gets inconsistent and so does y_2. This is a *redundant invocation*, i.e. a method on a replica is invoked multiple times by multiple instances of a method.

Since every method is deterministic, the same computation of the method t is performed on the replicas x_1 and x_2. Here, t_1 and t_2 are referred to as *same crone* instances of the method t. Instances u_{11}, u_{12}, u_{21}, and u_{22} are also same crones of the method u.

[**Definition**] A pair of instances t_1 and t_2 of a method t are same crones iff t_1 and t_2 are invoked on a replica by a same instance or by same crones. □

A quorum of an object x for a method t is *expanded* in a transaction T iff same crone instances of t invoked in T are performed on more number of replicas of x than the quorum number N_t. An instance t is *redundantly invoked* on a replica iff a same crone as t is already invoked on the replica. In order to resolve the quorum expansion and redundant invocation, each instance issued to a replica is required to satisfy following constraints:

[**Invocation constraints**]

1. $Q_{ui} = Q_{uj}$ for every pair of same crones u_i and u_j issued from replicas x_i and x_j, respectively.
2. At most one crone instance of a method invoked in a transaction is performed on each replica if the method is a dependent or update type. □

[**Theorem**] If every method is invoked on a replica so that the invocation constraint is satisfied, neither quorum expansion nor redundant invocation occurs. □

4. Redundant Invocation

4.1 Basic Protocol

In order to resolve the redundant invocation, we have to make clear whether or not every pair of instances issued to a replica are same crones. An identifier $id(t_i)$ for each instance t_i invoked on a replica of an object x is composed of a method type t and identifier of the

object x, i.e. $id(t_i) = t{:}x$. Each transaction T has a unique identifier $tid(T)$, e.g. thread identifier. If the transaction T invokes a method t, t is assigned a transaction identifier $tid(t)$ as a concatenation of $tid(T)$ and *invocation sequence number* $iseq(T, t)$ of t in T. The invocation sequence number is incremented by one each time T invokes a method. Thus, $iseq(T, t)$ shows how many methods T has invoked before invoking t_i. Suppose an instance t_i on a replica x_i invokes an instance u_k on a replica y_k. $id(t_i) = t{:}x$. The transaction identifier $tid(u_k)$ is $tid(t_i){:}id(t_i){:}iseq(t_i, u_k) = tid(t_i){:}t{:}x{:}iseq(t_i, u_k)$. $id(u_k) = u{:}k$. Thus, $tid(u_k)$ shows an invocation sequence of methods from T to the instance u_k. The transaction identifiers have to satisfy the following constraint.

[Transaction identifier] $tid(t_1) = tid(t_2)$ iff t_1 and t_2 are same crone instances. □

Suppose $tid(T)$ is assumed to be 6 in Fig. 4. Suppose T invokes a method t after invoking three methods, i.e. $iseq(T, t_1) = iseq(T, t_2) = 4$. $id(t_1) = id(t_2) = t{:}x$. Since $tid(t_1) = tid(t_2) = tid(T){:}iseq(T, t_1) = tid(T){:}iseq(T, t_2) = 6{:}4$, t_1 and t_2 are same crone instances. The method t invokes another method u after invoking one method. Here, $iseq(t,u)=2$, $tid(u_{11}) = tid(u_{12}) = tid(t_1){:}id(t_1){:}2 = 6{:}4{:}t{:}x{:}2$. $tid(u_{21}) = tid(u_{22}) = tid(t_2){:}id(t_2){:}2 = 6{:}4{:}t{:}x{:}2$. Since $tid(u_{11}) = tid(u_{21})$, u_{11} and u_{21} are same crone instances on a replica y_1.

A method t invoked on a replica x_h is performed as follows:

1. If no method is issued to a replica x_h, an instance t_h is performed and a response res of t is sent back. $\langle t, res, tid(t_h)\rangle$ is stored in the log L_h.
2. If $\langle t, res, tid(t'_h)\rangle$ such that $tid(t_h) = tid(t'_h)$ is found in L_h, the response res of t'_h is sent back as the response of t_h without performing t_h. Otherwise, t is performed on the replica x_h as presented at step 1.

In Fig. 4, suppose u_{11} is first issued to the replica y_1. $\langle u$, response of u_{11}, $tid(u_{11})\rangle$ is stored in the log L_1. Then, u_{21} is issued. Since $tid(u_{11}) = tid(u_{21})$, i.e. u_{11} and u_{21} are same crones, u_{21} is not performed but the response of u_{11} stored in the log L_1 is sent to t_2 as the response of u_{21}. By the resolution of the redundant invocation, at most one crone instance is surely performed on each replica. In addition, if multiple instances invoke a same method on a replica, every invoker instance receives the same response of the method.

4.2 Modified Protocol

At the deeper level methods are invoked, the longer the length of the transaction identifier is getting. We try to reduce the length of the transaction identifier. Suppose a transaction T invokes a method u on an object y in addition to invoking a method t as shown in Fig. 1. The method t invokes the method u as well. If the transaction identifier $tid(T)$ is used as an identifier of each method, both instances of u invoked by T and t have the same identifier. Hence, if an instance of u invoked by T is already performed on a replica of y, other instances of u invoked by t are not performed. As long as every method is invoked at most once in a transaction T, the transaction identifier $tid(T)$ can be used as an identifier of each instance.

Next, suppose every transaction and method serially issue methods. Each transaction T has a variable id whose initially value is 0. Suppose a transaction T issues a method t on replicas $x_1, \ldots, x_m$. Each request message carries the method t with $tid(T)$ and id. Then, an instance t_i of the method t is performed on a replica x_i. Suppose t_i issues a method u to replicas $y_1, \ldots, y_l$. id is incremented by one, $id := id + 1$. The request u with $tid(T)$ and id is carried to replicas of the object y. If an instance u_j of the method u finishes on a replica y_j, the response of u_j carries id to the invoker instance t_u. Then, the value of id in t_i is replaced with id returned from u_j. Then, t_i issues another method v on replicas of an object z. id is incremented by one. Then, $tid(T)$ and id are sent to the replicas of z. Thus, id shows a depth-first order of methods in an invocation tree of methods.

1 On receipt of a request $\langle t, tid(T), id\rangle$ from an invoker instance s,
   a. $cid := sid := id$; t is performed;
   b. If t invokes a method u on an object y, $cid := cid + 1$ and a request $\langle u, tid(T), cid\rangle$ is issued to replicas of y. t waits for a response from the replicas.

2 On receipt of a response $\langle u, tid(T), id, resp\rangle$
   1. $cid := id$;
   2. If t invokes another method, go to 1b.
   3. A response $\langle t, tid(T), cid, resp'\rangle$ is sent to the instance where $resp'$ is the response of t.

The transaction identifier of each instance in-

voked in a transaction T is a pair $\langle tid(T), id\rangle$. If the method t finished on a replica x_i, $\langle t, tid(T), sid, resp\rangle$ is stored in the log L_i where *resp* shows response data of t. On receipt of a request $\langle t, tid, id\rangle$, L_i is searched. If $\langle t, tid, id, resp\rangle$ is found in L_i, the request is not performed because a same crone is already performed on x_i. Without performing t, the response $\langle t, tid, id, resp\rangle$ is sent to the invoker.

[Property] $tid(t_1) = tid(t_2)$ iff t_1 and t_2 are same crones. □

5. Quorum Expansion

5.1 Basic Protocol

Suppose a method t on an object x invokes a method u on an object y. Let Q_{uh} be a quorum of u invoked by an instance t_h of the method t on a replica x_h. In order to resolve the quorum expansion, Q_{uh} and Q_{uk} have to be the same for every pair of replicas x_h and x_k. If some method is frequently invoked, the replicas in the quorum are overloaded. The quorum of the method u has to be randomly decided each time u is invoked. In distributed systems, the quorum information is distributed in networks. If some replica is faulty, the quorum including the faulty replica has to be updated. We have to discuss a mechanism to randomly create a quorum Q_{ui} for each invoker instance t_i to invoke a method u in presence of replica fault of y, which satisfies the following constraints:

1. $Q_{ui} = Q_{uj}$ only if a pair of instances t_i and t_j are same crones in a transaction.
2. $Q_{ui} \neq Q_{uj}$ if t_i and t_j are different crones.

We introduce a function *select*(i, n, a) which gives a set of n numbers out of 1, ..., a for a same initial value i where $n \leq a$. For example, *select*$(i, n, a) = \{h \mid h = (i + \lceil \frac{a}{n} \rceil (j - 1))$ *modulo* a for $j = 1, \ldots, n\} \subseteq \{1, \ldots, a\}$. For a pair of different values x and y, *select*$(x, n, a) \neq$ *select*(y, n, a). By using *select*, an instance t_h on a replica x_t obtains a quorum Q_{uh} of a method u as follows: Suppose an instance t_h on a replica x_h invokes a method u. I = *select*$(numb(tid(t_h)), N_u, b)$ is obtained, where N_u is quorum number of u and b is a total number of replicas of y, i.e. $\{y_1, \ldots, y_b\}$. Let $tid(t_h)$ be $s_1{:}s_2{:}\cdots{:}s_g$. Here, $numb(tid(t_h))$ is $(s_1 + \cdots + s_g)$ *modulo* a. $I \subseteq \{1, \ldots, b\}$ and $|I| = N_u$. Then, $Q_{uh} = \{y_i \mid i \in I\}$.

Every pair of same crone instances have the same transaction identifier *tid* as presented in the preceding Subsection. Hence,

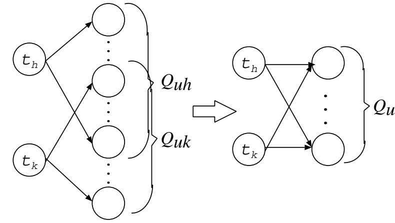

Fig. 5 Resolution of quorum expansion.

select$(numb(tid(t_h)), N_u, b)$ = *select*$(numb(tid(t_k)), N_u, b)$ for every pair of crone instances t_h and t_k. An instance t_h on every replica x_h issues a method u to the same quorum Q_{uh} as the other same crones. Hence, no quorum expansion occurs (**Fig. 5**). In addition, a quorum Q'_{uk} obtained for another crone instance t'_h is different from Q_{uh}.

5.2 Modified Protocol

Each instance t_h on a replica x_h issues a method request u to N_u replicas of the object y. Hence, totally $N_t \cdot N_u$ requests are transmitted. We try to reduce the number of requests transmitted in the network. Let Q_u be a quorum $\{y_1, \ldots, y_b\}$ $(b = N_u)$ of the method u obtained by the function *select* for each instance t_h. If each instance t_h issues a request u to only a subset $Q_{uh} \subseteq Q_u$, the number of requests issued to the replicas of the object y can be reduced. Here, $Q_{u1} \cup \ldots \cup Q_{ua} = Q_u$.

Let r (≥ 1) be a *redundancy factor*, i.e. the number of the requests to be issued to each replica y_k in Q_u. For each instance t_h on a replica x_h in $Q_t = \{x_1, \ldots x_a\}$ where $a = N_t$, Q_{uh} is constructed for the method u as follows $(h = 1, \ldots, a)$:

If $a \geq b{\cdot}r$, $Q_{uh} = \{y_k \mid k = \lceil \frac{hb}{a} \rceil\}$ if $h \leq r \cdot b$ $Q_{uh} = \phi$ *otherwise*.

If $a < b{\cdot}r$, $Q_{uh} = \{ y_k \mid (1 + \lfloor \frac{(h-1)b}{a} \rfloor) \leq k < [\, 1 + (\, \lfloor \frac{(h+r-1)b}{a} \rfloor - 1)$ *modulo* $b\,]\}$.

For example, suppose instances t_1, t_2, and t_3 on replicas x_1, x_2, and x_3, respectively, issue a method request u to replicas y_1, y_2, y_3, and y_4, i.e. $Q_t = \{x_1, x_2, x_3\}$ and $Q_u = \{y_1, y_2, y_3, y_4\}$. Suppose the redundancy factor $r = 2$. Hence, $Q_{uh} = \{y_k \mid (1 + (\lfloor \frac{(h-1)4}{3} \rfloor) \leq k \leq (1 + (\lfloor \frac{(h-1)4}{3} \rfloor + \lfloor \frac{8}{3} \rfloor - 1)$ *modulo* $4)\}$. Hence, $Q_{u1} = \{y_1, y_2\}$, $Q_{u2} = \{y_2, y_3, y_4\}$, and $Q_{u3} = \{y_3, y_4, y_1\}$ (**Fig. 6** (1)). Two requests from the instances of the method t are issued to each replica of y. For example, suppose an instance t_1 on a replica x_1 is faulty. Another instance t_2 sends u to the replicas y_2, y_3, and y_4 in Q_{u2} and

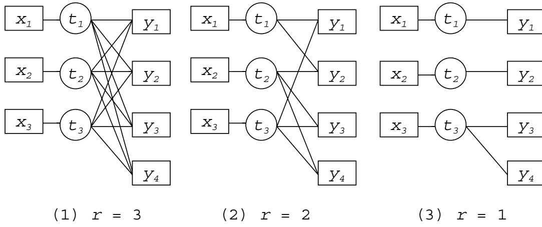

Fig. 6 Invocations.

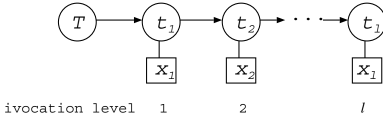

Fig. 7 Invocation model.

t_3 sends u to the replicas in Q_{u3}. Since $Q_{u2} \cup Q_{u3} = \{y_1, y_2, y_3, y_4\}$, u is sent to every replica in Q_u even if t_1 is faulty. $Q_{u1} = \{y_1\}$, $Q_{u2} = \{y_2\}$, and $Q_{u3} = \{y_3, y_4\}$ for $r = 1$ (Fig. 6 (3)). Thus, totally $r \cdot N_u$ requests of the method u are issued to the replicas in Q_u. Even if $(r-1)$ instances of t are faulty, u is performed on N_u replicas of y.

6. Evaluation

We evaluate the *quorum-based invocation* (Q) protocol discussed in this paper. The Q protocol is evaluated in terms of number of replicas manipulated, number of requests issued, and response time compared with the *primary − secondary invocation* (P) protocol.

In the evaluation, we take a simple invocation model where a transaction T first invokes a method t_1 on an object x_1, then t_1 invokes t_2 on x_2, $\cdots$ as shown in **Fig. 7**. Here, let a_i be the number of replicas of an object x_i ($i = 1, 2, \ldots$). Let N_i be the quorum number of a method t_i ($N_i \leq a_i$), where i shows a level of invocation. Let r_i be a *redundancy factor* on an object x_i. In the primary-secondary (P) protocol, only a method on a primary replica is invoked as shown in Fig. 2. Suppose a method t_i invokes another method t_{i+1} on a primary replica x_{i+1} (**Fig. 8**). If x_{i+1} is faulty, one secondary replica x'_{i+1} is taken as a new primary and a method t_{i+1} on the replica x_{i+1} is invoked again. In addition, x'_{i+1} might be faulty during invocation of t_{i+1}. Here, if x'_{i+1} is detected to be faulty, another replica x''_{i+1} is taken and t_{i+1} is invoked again on the replica x''_{i+1}. Let f_i be probability that a replica of an object x_i is faulty. Thus, the higher f_i is, the longer it takes to perform the transaction T. We assume

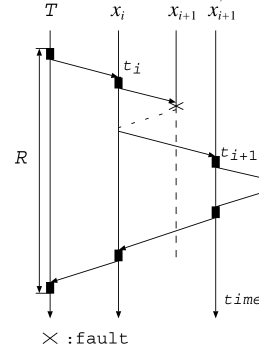

Fig. 8 Primary-secondary (P) protocol.

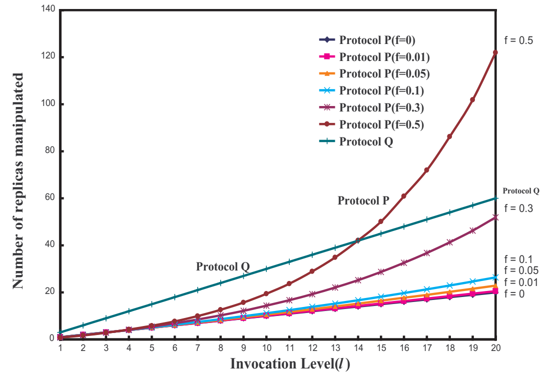

Fig. 9 Number of replicas manipulated.

$f_1 = f_2 = \ldots = f$.

In the Q protocol, each method t_i is performed on only N_i replicas of an object x_i as long as at least r_i replicas are operational. We assume that $a_1 = a_2 = \ldots = a = 10$, $N_1 = N_2 = \ldots = N$ $(\leq a)$, and $r_1 = r_2 = \ldots = r$.

Figure 9 shows the number of replicas where methods are performed in the transaction whose maximum invocation level is i for fault probability f. In the Q protocol, $a = 10$ and $N = 3$. Only the quorum number N of replicas, i.e. three replicas, in ten replicas are manipulated at each invocation level.

Figure 10 shows the number of request messages transmitted for fault probability f. In the Q protocol, N messages are transmitted. We assume the redundancy factor $r = N$ in this evaluation. N^2 request messages are transmitted at each invocation. Hence, $N^2 i$ request messages are transmitted for a transaction with invocation level i. The numbers of replicas manipulated are shown for $r = N$ and $r = N/3$. In the P protocol, totally i request messages are transmitted if no fault occurs, i.e., $f = 0$.

Let us consider response time of transaction

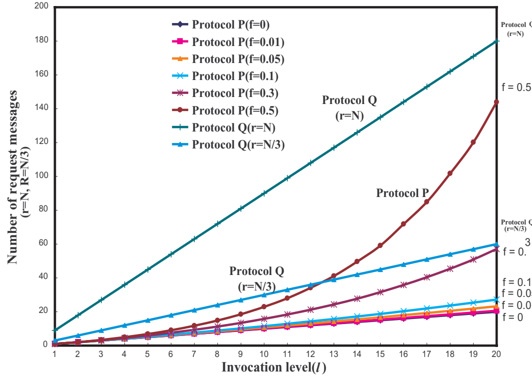

Fig. 10 Number of request messages issued.

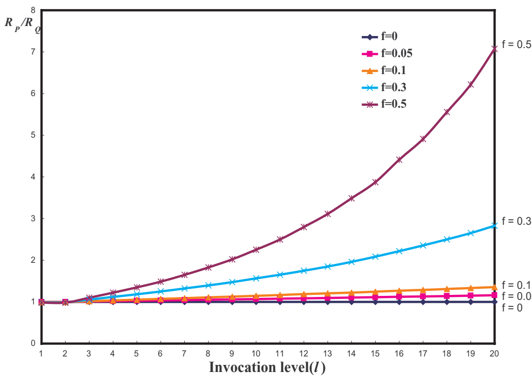

Fig. 11 Response time (α = 0.25).

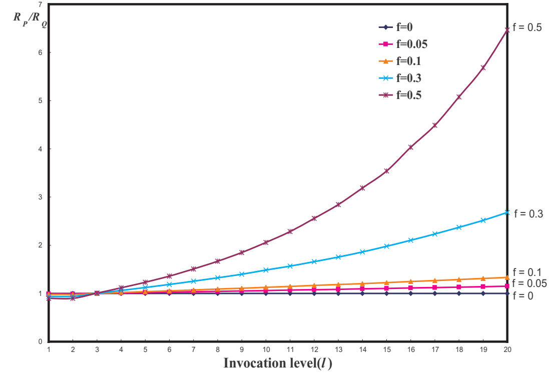

Fig. 12 Response time (α = 4).

with invocation level i in the Q and P protocols. Let δ_i be delay time to send a message from a replica of x_{i-1} to a replica of x_i. Let π_i show time for processing a request on a replica x_i. Here, we assume $\delta_1 = \delta_2 = \ldots = \delta$ and $\pi_1 = \pi_2 = \ldots = \pi$. In the Q protocol, the response time R_Q is $(2\delta + \pi)i$. In the P protocol, the response time R_P is $2\delta\cdot$(number of request messages) + $\pi\cdot$(number of replicas manipulated) for fault probability f, which are obtained from Figs. 9 and 10. Here, $\pi = \alpha\cdot\delta$. **Figures. 11** and **12** show the ratio R_P/R_Q for α=0.25 and α=4. α=0.25 shows the delay time is for times longer than the primary speed. These figures show that the Q protocol supports shorter response time than the protocol while implying larger number of messages transmitted.

7. Concluding Remarks

In this paper, we discussed how transactions invoke methods on multiple replicas of objects in a nested manner. Methods may invoke other methods, i.e. nested invocation. If methods are invoked on multiple replicas, multiple redundant instances of a same method may be performed on a replica, *redundant invocation* and more number of replicas than the quorum number may be manipulated, *quorumn expansion*. We discussed the Q (quorum-based invocation) protocol with neither redundant invocations nor quorum expansions. We evaluated the Q protocol compared with primary-secondary one. We showed the Q protocol implies shorter response time while more number requests are transmitted than the primary-secondary one. By using the Q protocol, a replicated object-based system can be efficiently realized.

References

1) Barrett, P.A., Hilborne, A.M., Bond, P.G. and Seaton, D.T.: The Delta-4 Extra Performance Architecture, *Proc. 20th Int'l Symp. on FTCS*, pp.481–488 (1990).

2) Bernstein, P.A. and Goodman, N.: The Failure and Recovery problem for Replicated Databases, *Proc. 2nd ACM POCS*, pp.114–122 (1983).

3) Budhiraja, N., Marzullo, K., Schneider, B. and Toueg, S.: *The Primary-Backup Approach*, ACM Press, pp.199–221 (1984).

4) Carey, J.M. and Livny, M.: Conflict Detection Tradeoffs for Replicated Data, *Proc. ACM TODS*, Vol.16, No.4, pp.703–746 (1991).

5) Chevalier, P.Y.: A Replicated Object Server for a Distributed Object-Oriented System, *Proc. IEEE SRDS*, pp.4–11 (1992).

6) Garcia-Molina, H. and Barbara, D.: How to Assign Votes in a Distributed System, *JACM*, Vol.32, No.4, pp.841–860 (1985).

7) Gifford, D.K.: Weighted Voting for Replicated Data, *Proc. 7th ACM Symp. on Operating Systems Principles*, pp.150–159 (1979).

8) Koo, R. and Toueg, S.: Checkpointing and

Rollback-Recovery for Distributed Systems, *IEEE Trans. Softw. Eng.*, Vol.3E-13, No.1, pp.23–31 (1987).
9) Schneider, B.F.: Replication Management using the State-Machine Approach, *Distributed Computing Systems*, Vol.7, ACM Press, 2nd edition (1993).
10) Silvano, M. and Douglas, C.S.: Constructing Reliable Distributed Communication Systems with CORBA, *IEEE Comm. Magazine*, Vol.35, No.2, pp.56–60 (1997).
11) Tanaka, K., Hasegawa, K. and Takizawa, M.: Quorum-Based Replication in Object-Based Systems, *Journal of Information Science and Engineering*, Vol.16, pp.317–331 (2000).
12) Wiesmann, M., et al.: Understanding Replication in Databases and Distributed Systems, *Proc. IEEE ICDCS-2000*, pp.264–274 (2000).

(Received June 27, 2002)
(Accepted October 7, 2002)

Kenichi Hori was born in 1979. He received his B.E. degrees in Computers and Systems Engineering from Tokyo Denki University, Japan in 2002. He is now a graduate student of the master course in the Department of Computers and Systems Engineering, Tokyo Denki University. His research interests include distributed database systems and fault-tolerant system. He is a student member of IPSJ.

Tomoya Enokido was born in 1974. B.E. and M.E. degrees in Computers and Systems Engineering from Tokyo Denki University, Japan 1997 and 1999. After he worked for NTT Data Corporation, he is currently a research assistant in the Department of Computers and Systems Engineering, Tokyo Denki University. His research interests include distributed systems and group communication. He is a member of IPSJ.

Makoto Takizawa served as program co-chair of the IEEE International Conference on Distributed Computing Systems (ICDCS) in 1998 and as program vice chairs of ICDCS in 1994 and 2000, and is serving as a general co-chair of ICDCS-2002. He also served as a general co-chair of IEEE ISORC. He is a member of the program committees of many IEEE Computer Society conferences including ICDCS, SRDS, ICPADS, ISORC, and ICNP. He was elected for 2003–2005 BoG member of IEEE Computer Society. Takizawa is a full professor in the Department of Computers and Systems Engineering, Tokyo Denki University, Japan. He is now a dean of the graduate school of Science and Engineering, Tokyo Denki University. He chaired the Information Division at the Research Institute for Technology, Tokyo Denki University from 1998 to 2002. He was a visiting professor at GMD-IPSI, Germany (1989–1990) and has been a regular visiting professor at Keele University, England since 1990. Takizawa is a fellow of Information Processing Society of Japan (IPSJ) and was a member of the executive board of IPSJ from 1998 to 2000. He chaired SIGDPS (distributed processing) of IPSJ from 1997 to 2000 and was an editor of the Journals of IPSJ (1994–1998). Takizawa received his BE, ME, and DE in computer science from Tohoku University, Japan. In 1996, he won the best paper award at IEEE International Conference on Parallel and Distributed Systems (ICPADS). He is a member of the IEEE and a member of the ACM and IPSJ.