マルチメディア通信と分散処理 99－2
（2000. 9. 22）

Quorum-based Locking Protocol for Replicated Objects

Katsuya Tanaka and Makoto Takizawa
Tokyo Denki University
Email {katsu, taki}@takilab.k.dendai.ac.jp

In object-based systems, objects are encapsulations of data and procedures named methods. We discuss how to lock replicated objects, where objects support high-level methods and methods are invoked in a nested manner, by extending the quorum-based scheme. If a pair of methods t and u are compatible, each method is surely performed on some replica but both of them may not be performed on a same replica in our protocol. Compatible methods are exchanged between the replicas if both the methods are not performed on any of the replicas. If a method t is invoked on multiple replicas and each instance of t invokes another method u, u is performed multiple times on an object, i.e. redundant invocation. In addition, since each instance of t issues a request u to its quorum, more number of the replicas are locked than the quorum number of t. This is quorum explosion. We discuss how to resolve these redundant invocations and quorum explosion.

多重化されたオブジェクト間の一貫性を保証するレプリカロック手法

田中 勝也　滝沢 誠
東京電機大学
E-mail {katsu, taki}@takilab.k.dendai.ac.jp

オブジェクトは、データ構造とメソッドから構成され、データは、メソッドを介して操作される。メソッドの実行要求の受信において、オブジェクトは、要求されたメソッドを起動し、結果を返す。さらに、起動されたメソッドは、他のオブジェクトのメソッドを起動する入れ子型となる。システム全体の信頼性や可用性の向上を目的として、システム内の各オブジェクトは、複数のレプリカ (コピー) に多重化される。本研究では、演算間の互換関係に基づく、レプリカ間の一貫性を保証するロック手法を提案する。さらに、入れ子型演算によって引き起こされる問題を明確にし、その解決方を示す。

1 Introduction

Distributed applications are realized in object-based frameworks like CORBA [10]. In order to increase the reliability, availability, and performance, objects are replicated. In the two-phase locking (2PL) protocol [1, 3], one of the replicas for *read* and all the replicas for *write* are locked. In the quorum-based protocol [5], *quorum* numbers N_r and N_w of the replicas are locked for *read* and *write*, respectively. The subset of the replicas is a *quorum*. Here, a constraint "$N_r + N_w > a$" for the number a of the replicas has to be satisfied. An object is manipulated only through *methods*. A pair of methods t and u of an object o conflict if the result obtained by performing t and u depends on the computation order. Before performing t, a *quorum* number N_t of the replicas of an object o are locked. Suppose a pair of methods t and u are issued to the replicas. The method t may be performed on one replica o_t and u on the other o_u if t and u are compatible. Here, the state of o_t is different from o_u if t or u is an update method. o_t and o_u can be the same if each method performed on a replica is performed on the other replica, i.e. u and t are performed on o_t and o_u, respectively. Thus, the newest version can be constructed from replicas by exchanging methods performed. Thus, "$N_t + N_u > a$" only if t and u conflict on an object o. Even if t or u updates the object o, i.e. *write*, $N_t + N_u \leq a$ if t and u are compatible. The authors [11] discuss a version vector to identify which methods are performed on each replica. The method implies larger overhead. In this paper, we discuss a simpler method to exchange methods among replicas.

In the object-based system, methods are invoked in a nested manner. Suppose a method t on an object x invokes a method u on another object y. x is replicated in replicas x_1 and x_2 and y is replicated in y_1 and y_2. A method t is issued to x_1 and x_2. Then, the method t invokes u on y_1 and y_2. Here, u is performed twice on each replica. If u updates y, y is inconsistent. This is a *redundant invocation*. The method u should be performed just once on each of y_1 and y_2. In addition, an instance of t on x_1 issues u to its own quorum, say Q_1, and t on x_2 issues to Q_2 where $|Q_1| = |Q_2| = N_u$. The replicas in $Q_1 \cup Q_2$ are locked by the method t. Since $|Q_1 \cup Q_2|$ is larger than the quorum number N_u, more number of replicas are locked than the quorum number N_u. This is a *quorum explosion*. We discuss how to resolve the redundant invocations and quorum explosions in nested invocations of methods on replicas.

In section 2, we present a system model. In section 3, we extend the quorum concept on *read* and *write* to the object-based system. In section 4, we discuss how to resolve the redundant invocation and quorum explosion. In section 5, we present the evaluation.

2 System Model

2.1 Replicas

A system is composed of replicas of objects. The replicas are distributed in multiple computers. Each object supports a collection of methods only by which the object is manipulated. A transaction issues a method request *op* to an object o. Then, *op* is performed on the object o and the response of *op* is sent back to the transaction.

Here, op is referred to as *invoked*. A method op_t is *compatible* with op_u iff the result obtained by performing op_t and op_u on o is independent of the computation order of op_t and op_u. Otherwise, op_t *conflicts* with op_u. The conflicting relation is symmetric but not transitive. The method op performed on o may furthermore issue a request to another object. Thus, methods are invoked on objects in a nested manner. In this paper, we assume each of transactions and methods invokes one method at a time.

Suppose there are two objects x and y in the system. There are three replicas, x_1, x_2, and x_3 for the object x and two replicas y_1 and y_2 for y. The object x supports a method t which invokes a method u on the object y. A transaction issues a request t to replicas of x. First, we consider a quorum Q_t, i.e. to which replica out of x_1, x_2, and x_3 the request t is issued. In the famous two-phase locking (2PL) protocol, a transaction T issues a *write* request to all the replicas, x_1, x_2, and x_3 but a *read* request to one replica, say x_1. The *read* request does not change the state of the object but the *write* request changes. In the quorum-based protocol, T issues *write* and *read* requests to some numbers N_w and N_r of replicas of x, respectively. Here, $N_w + N_r > a$ where a is the total number of the replicas of x, i.e. $a = 3$. For example, a *write* request is issued to a subset $\{x_1, x_2\}$ denoted a *write* quorum Q_w and a *read* request is issued to a subset $\{x_2, x_3\}$ denoted a read quorum Q_r. $N_w\ (= |Q_w|) = N_r\ (= |Q_r|) = 2$. The *read* and *write* methods are surely performed on the replica x_2 in $Q_r \cap Q_w$ while only *write* and *read* are performed on x_1 and x_3, respectively. In this paper, we discuss to which replicas each request is issued. An object supports procedures as *methods*. We extend the traditional quorum-based locking protocol which is used for *read* and *write* on a simple object to objects which support methods to be invoked in the nested manner.

2.2 Nested invocation

Methods are invoked in a nested manner. Suppose a transaction T issues a request t to a quorum $Q_t = \{x_1, x_2, x_3\}$. The method t is performed on every replica in Q_t. In performing t on each replica of x, a request u is invoked on all the replicas y_1 and y_2. Each of three method instances on x_1, x_2, and x_3 invokes the method u on y_1 and y_2 [Figure 1]. Hence, the method u is performed three times on each replica of y. If u updates y, the states of the replicas y_1 and y_2 get inconsistent. For example, u is a method which increments y by one. If the object x is not replicated, the value of y is just incremented by one. However, the value of y is incremented by three since x and y are replicated as presented here. Even if u does not update y, the same computation which outputs the same result is performed three times. It consumes computation resource. This is a *redundant invocation*.

In Figure 1, each instance of the method x issues request u to the same quorum $Q_u = \{y_1, y_2\}$. If each instance x_i decides its own quorum Q_{ui}, $|Q_{u1} \cup Q_{u2} \cup Q_{u3}| \geq |Q_u|$. That is, more number of replicas of y are locked by the transaction T than Q_u. This is a *quorum explosion*. We have to resolve the redundant invocation and quorum explosion to occur in the nested invocation of methods on replicas.

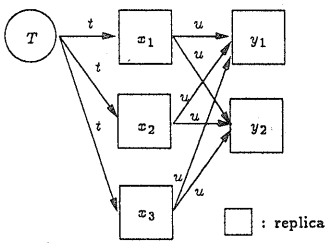

Figure 1: Redundant invocation.

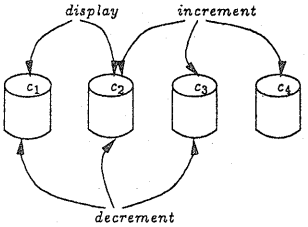

Figure 2: Quorum.

2.3 Extension of quorum

Let us consider a *counter* object c which supports three methods *increment* (*inc*), *decrement* (*dec*), and *display* (*dsp*). Suppose there are four replicas c_1, c_2, c_3, and c_4 of the *counter* object, i.e. $R_c = \{c_1, c_2, c_3, c_4\}$. According to the traditional quorum-based theory, *increment* and *decrement* are considered to be update methods, i.e. *write* ones. Hence, $N_{inc} + N_{dec} > 4$, $N_{dsp} + N_{inc} > 4$, and $N_{dsp} + N_{dec} > 4$. For example, $N_{inc} = N_{dec} = 3$ and $N_{dsp} = 2$ [Figure 2]. The methods *display* and *increment* conflict. $N_{dsp} + N_{inc}$ is required to be larger than 4 [Figure 3]. That is, both *display* and *increment* are performed on at least one replica. The methods *increment* and *decrement* are compatible on the *counter* object because the state obtained by performing *increment* and *decrement* is independent of the computation order of the methods. Suppose *increment* is issued to two replicas c_1 and c_2 and *decrement* is issued to c_3 and c_4. Since both *increment* and *decrement* are performed not on any replica, i.e. either *increment* or *decrement* is performed, the states of c_1 and c_3 are different. However, if *decrement* is performed on c_1 and c_2 and *increment* is performed on c_3 and c_4 here, the states of c_1, c_2, c_3, and c_4 can be the same. This is referred to as *exchanging procedure* where methods performed on one replica is sent to other replicas where the methods are not performed and only methods compatible with the methods are performed. As long as only *increment* and *decrement* are issued to the replicas, the exchanging procedure is not required to

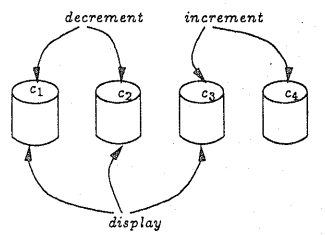

Figure 3: Object-based quorum.

be executed. Suppose a *display* method is issued to three replicas c_1, c_2, and c_3 where $N_{dsp} = 3$. *display* conflicts with *increment* and *decrement*. The method *display* cannot be performed on any replica of c_1, c_2, and c_3 because only *increment* is performed on c_1 and c_2 and *decrement* on c_3 [Figure 4]. Before performing *display*, *decrement* has to be performed on c_1 and c_2 and *increment* on c_3. *increment* and *decrement* can be performed in any order because they are compatible. Here, c_1, c_2, and c_3 get the same, $c_1 = c_2 = c_3$ because both *increment* and *decrement* are performed on every replica. Then, *display* can be performed on c_1, c_2, and c_3. We discuss a new quorum-based locking protocol with the exchanging procedure.

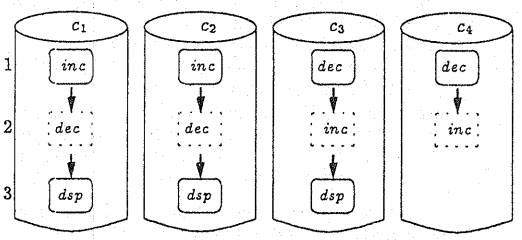

Figure 4: Exchanging procedure.

A *cluster* R is a set of replicas $o_1, \ldots, o_a$ of an object o ($a \geq 1$). Let Q_t be a *quorum*, i.e. subset of the replicas to be locked by a method t ($Q_t \subseteq R$). Let N_t ($= |Q_t|$) be the *quorum* number of t. The quorums of methods t and u of o have to satisfy the following constraint in the object-based systems.

[OBQ constraint]

- $N_t + N_u > a$ iff t conflicts with u. □

A transaction T invokes a method t on an object o. First, a quorum Q_t for t is constructed by selecting N_t replicas in the cluster R, for example randomly selecting replicas. If every replica in Q_t is locked, the replicas in Q_t are manipulated by t. In the quorum-based protocol, $N_t + N_u > a$ if a pair of methods t and u are update ones. On the other hand, the OBQ constraint means that $N_t + N_u > a$ only if t conflicts with u. It is noted that $N_t + N_u \leq a$ if t and u are compatible even if t or u updates o. The OBQ constraint implies the following properties:

[**Property**] Every pair of conflicting methods t and u of an object o are performed on at least k ($= N_t + N_u - a$) replicas in the same order. □

3 Exchanging Procedure

We discuss the exchanging procedure. Each replica o_h has a log L_h where a sequence of update methods performed on o_h are stored. We discuss how to manipulate the log L_h. Initially, L_h is empty. Suppose that a method *op* is issued to o_h. If *op* is an update method, *op* is stored in L_h, i.e. $L_h = \langle op \rangle$. Here, let L_h be a sequence of update methods $\langle op_{h1}, \ldots, op_{hm} \rangle$. Suppose a method *op* is issued to o_h. If *op* is compatible with every method op_{hi}, *op* is enqueued into L_h, i.e. $L_h = \langle op_{h1}, \ldots, op_{hm}, op \rangle$ and then *op* is performed on o_h. Thus, every pair of methods in L_h are compatible. Suppose that *op* conflicts with a method

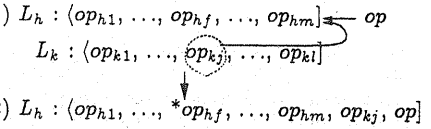

Figure 5: Exchanging procedure.

op_{hf} and *op* is compatible with every method op_{hg} ($g > f$) in L_h. There might be some replica o_k whose log L_k includes some method op_{kj} which is compatible with a method in L_h but conflicts with *op*, and is not performed on o_h. Such method op_{kj} is required to be performed on o_h before *op* is performed on o_h. Here, another replica o_k has a log $L_k = \langle op_{k1}, \ldots, op_{kl} \rangle$ and *op* is issued to o_k. *op* conflicts with op_{ku} and *op* is compatible with every method op_{kg} ($g > u$). According to the OBQ property, every pair of methods op_{hi} in L_h and op_{kj} in L_k are compatible. Here, a method op_{kj} in L_k is referred to as *missing method* for a method *op* on o_h iff op_{kj} is not performed on o_h and op_{kj} conflicts with *op* [Figure 5(1)]. Here, every missing method op_{kj} for *op* in L_k is required to be performed on o_h before *op* is performed. Then, *op* is performed on o_h. All the methods conflicting with *op* are marked ($*$) in L_h and *op* is enqueued into L_h [Figure 5(2)]. Every pair of unmarked methods in a log are compatible. If an update method *op* is marked in every log, *op* is performed on every replica and some conflicting method is performed after *op*. Hence, *op* is removed from every log.

Let us consider four replicas c_1, c_2, c_3, and c_4 of the *counter* object which support methods *inc* (*increment*), *dec* (*decrement*), and *dsp* (*display*). Here, $N_{inc} = N_{dec} = 2$ and $N_{dsp} = 3$ according to the OBQ constraint [Figure 3]. First, a method *increment* (inc_1) is issued to c_1 and c_2. Then, *decrement* (dec_1) is issued to c_3 and c_4. The methods are performed and enqueued into the logs. Here, $L_1 = L_2 = \langle inc_1 \rangle$ and $L_3 = L_4 = \langle dec_1 \rangle$. Next, *increment* ($inc_2$) is issued to c_2 and c_3. Since *increment* and *decrement* are compatible, inc_2 is performed on c_2 and c_3 and enqueued into the logs L_2 and L_3. Here, $L_1 = \langle inc_1 \rangle$, $L_2 = \langle inc_1, inc_2 \rangle$, $L_3 = \langle dec_1, inc_2 \rangle$, and $L_4 = \langle dec_1 \rangle$. Next, a method *display* (dsp_1) is issued to c_1, c_2, and c_4. dsp_1 conflicts with inc_1, inc_2, and dec_1. inc_2 and dec_1 are missing methods for dsp_1 in c_1. Hence, inc_2 and dec_1 are performed on c_1. dec_1 is performed on c_2 and inc_1 and inc_2 are performed on c_4. Here, c_1, c_2, and c_4 are the same. Then, dsp_1 is performed on c_1, c_2, and c_4. The methods inc_1, inc_2, and dec_1 are marked $*$ in the logs L_1, L_2, and L_4 because they conflict with dsp_1. Since dsp_1 does not change the state, dsp_1 is not enqueued into the logs, i.e. $L_1 = \langle *inc_1, *dec_1, *inc_2 \rangle$, $L_2 = \langle *inc_1, *inc_2, *dec_1 \rangle$, $L_4 = \langle *dec_1, *inc_1, *inc_2 \rangle$, and $L_3 = \langle dec_1, inc_2 \rangle$. Then, $display(dsp_2)$ is issued to c_2, c_3, and c_4. inc_1 is not performed on c_3 although inc_1 is performed on c_2 and c_4. Hence, inc_1 is performed on c_3. Here, $c_2 = c_3 = c_4$. Then, dsp_2 is performed on c_2, c_3, and c_4. Here, $L_2 = \langle *inc_1, *inc_2, *dec_1 \rangle$, $L_3 = \langle *dec_1, *inc_2, *inc_1 \rangle$, and $L_4 = \langle *dec_1, *inc_1, *inc_2 \rangle$. inc_1, inc_2, and dec_1 are marked in every log, i.e. some method conflicting with the methods are performed after the method. The method

inc_1, inc_2, and dec_1 are removed from the logs. Then, $L_1 = L_2 = L_3 = L_4 = \langle\ \rangle$ [Figure 6].

A transaction T issues a request op to the replicas in a quorum Q_{op}. Each log L_h is manipulated as follows:

1. A log L_h of a replica o_h is searched. If every method in L_h is compatible with op, op is enqueued into L_h and op is performed on o_h.
2. If there is a some method in L_h which conflicts with op, a log L_h is sent back to T.
3. T collects the logs from the replicas, i.e. $L = \cup\{L_h \mid o_h \in Q_h\}$. T sends a log $L_h' = \{op' \mid op' \in L - L_h$ and op' conflicts with $op\}$ to the replica o_h. A method in L_h' is performed on o_h. Then, op is performed on o_h. Every method conflicting with op in L_h is marked.

Each object has to decide whether or not each marked method in the log is performed on every other replica. Each method op brings information dst showing on which replica op is performed. dst is represented in a bit map $b_1 \cdots b_a$ where "$b_k = 1$" means that op is performed on a replica c_k. For example, *increment* (inc_1) is issued to the replicas c_1 and c_2. Here, dst of inc_1 is 1100. inc_1^{1100} shows that inc_1 is performed on c_1 and c_2. In the exchanging procedure, the transaction T receives inc_1^{1100} from c_1 and c_2 after issuing *display* (dsp_1) to c_1, c_2, and c_4. Then, T sends inc_1^{1101} to c_1, c_2, and c_4, i.e. 1101 shows c_1, c_2, and c_4. $*inc_1^{1101}$ is stored in the log. When a transaction issues another *display* (dsp_2) to c_1, c_2, and c_3, c_3 sends inc_1^{0011} to the transaction T and c_1 and c_2 send $*inc_1^{1101}$. Here, $0011 \cup 1101 = 1111$. Hence, T sends inc_1^{1111} to every replica. Then, $*inc_1$ is removed from the log in every replica.

4 Nested Invocation

4.1 Redundant invocation

In the object-based system, methods are invoked in a nested manner. Suppose there are two objects x and y and a method t of x invokes a method u of y. Suppose that there are replicas $x_1, \ldots, x_a$ of the object x and replicas $y_1, \ldots, y_b$ of the object y.

A transaction T issues a method t to replicas in the quorum Q_t, say $N_t = 2$. Suppose t is issued to replicas x_1 and x_2. Furthermore, t issues a request to replicas in the quorum of y to invoke a method u. Here, suppose $N_u = 2$. Let t_1 and t_2 be instances of the method t performed on replicas x_1 and x_2, respectively. Each of t_1 and t_2 issues a request of the method u to replicas in a quorum of t. Here, let Q_{u1} and Q_{u2} be quorums of u for the instances t_1 and t_2, respectively. Suppose $Q_{u1} = Q_{u2} = \{y_1, y_2\}$. t_1 and t_2 issue u to both y_1 and y_2. Here, let u_{i1} and u_{i2} be instances of the method u performed on replicas y_1 and y_2, which are issued by the instance t_i (i = 1, 2), respectively. If u updates y, a state of y_1 is inconsistent because two instances u_{11} and u_{21} of the method u from t_1 and t_2 are performed on y_1 [Figure 7]. This is a *redundant invocation*, i.e. a method on a replica is invoked by multiple instances of a method invoked by a same instance.

In order to resolve the redundant invocation, the following strategies are adopted:

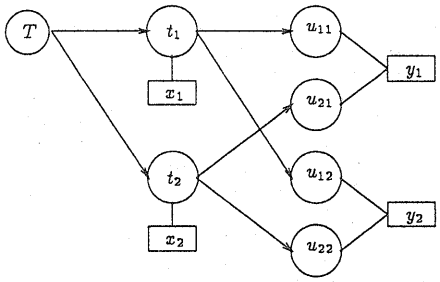

Figure 7: Redundant invocation.

1. Each transaction T is identified by a unique transaction identifier tid. Each request issued in T carries the tid of T.
2. If a method t is performed on a replica x, t and the response with tid of the transaction issuing t is logged into the log L_x.
3. Suppose a method t is issued to x. If t issued by the same transaction T is found in the log L_x of x, the response of t stored in the log is sent back without performing t.

By the redundant invocation resolution presented above, at most one instance of a method op is surely performed on each replica even if instances on multiple replicas invoke the method op. Here, suppose that there is another object z which supports a method v and that v invokes the method u on the object y. Suppose the transaction T invokes the method v on the object z in addition to invoking the method t on the object u. v also invokes u on y, say on a replica y_1. An instance t_1 is already performed on y_1. Since the tid of v is the same as t, the method u invoked by v is not performed according to the protocol presented here. In order to resolve this problem, each method instance op invoked on an object o has a unique identifier ($id(op)$) in the system. For example, $id(op)$ can be composed of the local identifier of op, e.g. thread identifier of op, a method type of op, and the identifier of o. Each transaction T has a unique transaction identifier $tid(T) = tid$ of T. If T invokes a method op, op is assigned a transaction identifier $tid(op) = id(T)$. Here, suppose a method op_2 on an object o_2 is invoked by a method instance op_1 on o_1. The transaction identifier of op_2, $tid(op_2)$, is given as a concatination of $id(op_1)$ and $tid(op_1)$, i.e. $tid(op_2) := tid(op_1).id(op_1)$. The identifier $tid(op)$ shows a invocation sequence of method instances from the transaction T to op. Suppose op is invoked on a replica o_h. op is performed on o_h as follows:

1. If op is performed on o_h, $\langle op$, response of op, $tid(op)\rangle$ is stored in the log L_h of o_h.
2. If a method instance op' such that $tid(op) = tid(op')$ is found in the log L_h, i.e. op and op' are instances of a same method and are invoked by a same instance, the response of op is sent back without performing op.

The redundant invocation is resolved by this method.

4.2 Quorum explosion

Suppose $Q_{u1} \neq Q_{u2}$, say $Q_{u1} = \{y_1, y_2\}$ and $Q_{u2} = \{y_2, y_3\}$ in the example of Figure 7. The method u is performed on replicas in $Q = Q_{u1} \cup$

	inc_1	dec_1	inc_2	dsp_1	dsp_2	
L_1	$\langle inc_1]$			$\langle *inc_1, *inc_2, *dec_1]$		$\langle\ ]$
L_2	$\langle inc_1]$		$\langle inc_1, inc_2]$	$\langle *inc_1, *inc_2, *dec_1]$		$\langle\ ]$
L_3		$\langle dec_1]$	$\langle dec_1, inc_2]$		$\langle *dec_1, *inc_2, *inc_1]$	$\langle\ ]$
L_4		$\langle dec_1]$		$\langle *dec_1, *inc_2, *inc_1]$		$\langle\ ]$

Figure 6: Example.

$Q_{u2} = \{y_1, y_2, y_3\}$. u is performed twice on the replicas in $Q_{u1} \cap Q_{u2} = \{y_2\}$ as presented here. If another transaction manipulates the object y by the method u, u is issued to the replicas in the quorum Q_u, say $\{y_3, y_4\}$. $|Q_{u1} \cup Q_{u2}| \geq |Q_u|$. This means that more replicas are locked than the quorum number N_u of the method u if the method u is invoked by a pair of the instances t_1 and t_2. Then, the instances of u on the replicas in $Q_{u1} \cup Q_{u2}$ issue further requests to other replicas and more replicas are locked. This problem is referred to as *quorum explosion*.

Suppose that a method t on an object x invokes a method u on an object y. Let Q_{uh} show a quorum of the method u invoked by an instance t_h of the method t on a replica x_h. In order to resolve the quorum explosion, Q_{uh} and Q_{uk} have to be the same for every pair of replicas x_h and x_k where an instance of the method t is performed. If a quorum Q_{uh} for t_h is constructed by a replica x_h independently of an instance t_k on every other replica x_k, the quorum explosion cannot be resolved. If a quorum is identical for every instance of a method t, i.e. $Q_{uh} = Q_{uk}$, only the same replicas are manipulated. If some replica is faulty, a quorum for every method including the faulty replica is required to be changed. We take an approach where a quorum is decided for each instance of a method.

There is a following approach to resolving the quorum explosion:

1. Each replica has a same function *rand* for generating a sequence of random numbers. That is, $rand(i, n, a)$ gives n random numbers from 1 to a for a same initial value i. For every replica, $rand(i, n, a)$ gives the same set of random numbers.
2. For each replica x_h, $I = rand(tid, N_h, a)$ is obtained, where tid is a transaction identifier of t, N_h is the quorum number of a method u, and a is a total number of replicas, i.e. $\{y_1, \ldots, y_a\}$. I is a set of replica numbers. Then, a quorum Q_h is constructed as $Q_h = \{y_i \mid i \in I\}$.

Every instance invoked by a same instance has the same transaction identifier as presented in the preceding subsection. An instance t_h of the method t on every replica x_h issues a request of u to the same quorum $Q_{uh} = Q_u$. Hence, the quorum explosion is prevented [Figure 8].

Another approach is that each instance t_h on a replica x_h decides a quorum Q_{uh} in order to invoke a method u on an object y. There are two cases, i.e. whether or not u conflicts with itself. First, suppose that u conflicts with itself. That is, $N_{uh} + N_{uk} > a$ for every pair of instances t_h and t_k where a is the total number of the replicas of y. Since $Q_{uh} \cap Q_{uk} \neq \phi$, t_h and t_k issues u to some replica o_l in $Q_{uh} \cap Q_{uk}$. The replica o_l decides which quorum Q_{uh} or Q_{uk} takes over the other. If Q_{uk} is taken over, the instance t_k stops issuing u to Q_{uk} and issues u to the replicas in Q_{uh}. Thus, every instance t_h can issue the method u to the same quorum. The other case is that u is compatible with itself, i.e. $Q_{uh} \cap Q_{uk}$ can be empty. If $Q_{uh} \cap Q_{uk} = \phi$, no replica receives requests from both t_h and t_k. Hence, each of t_h and t_k has to issue a request u to all the replicas in the quorum Q_{uh} and Q_{uk}, respectively. Then, suppose that u invokes another method v on an object z and v conflicts with itself. Each instance u_{uhi} in Q_{uh} issues a request z to a quorum Q_{uhi}. $Q_{uhi} \cap Q_{ukj} \neq \phi$ and $Q_{uhi} = Q_{uhl} \neq \phi$. A common replica in a pair of Q_{uhi} and Q_{ukj} can decide which quorum Q_{uhi} and Q_{ukj} is taken, say Q_{uhi} is taken if $id(u_{hi}) < id(u_{kj})$. Then, this decision is distributed to every replica in $Q_{uhi} \cup Q_{ukj}$. Suppose a replica z_p knows some decision, i.e. Q_z. If a request z with a quorum Q is issued to z_p. If $Q_z \neq Q$, one of Q_z and Q is taken by the same decision logic. Thus, if a method conflicting with itself is invoked, the quorum is convergent to the proper size of the quorum. However, as long as a method compatible with itself is invoked, the quorum can be exploded. Hence, we adopt the protocol presented here.

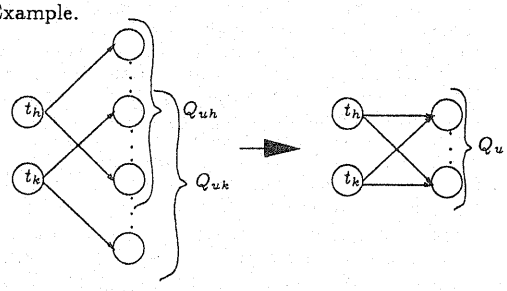

Figure 8: Resolution of quorum explosion.

5 Evaluation

We evaluate the protocol to resolve the redundant invocation and quorum explosion in terms of number of replicas locked and number of requests issued. The following three protocols R, Q, and N are considered:

1. Protocol R: without redundant invocation and quorum explosion.
2. Protocol Q: without redundant invocation.
3. Protocol N:

The protocol N supports no resolution of the redundant invocation and quorum explosion. In the protocol Q, the redundant invocation is prevented. In the protocol R, neither redundant invo-

cation nor quorum explosion occur. The protocol R shows our protocol discussed in this paper.

We consider a simple invocation model where a transaction first invokes a method t_1 on an object x_1, then t_1 invokes a method t_2 on an object x_2, $\cdots$. Here, let n_i be the number of replicas of an object x_i ($i = 1, 2, \ldots$). Let r_i be the quorum number of a method t_i ($r_i \leq n_i$). The number i shows a level of invocation. In the protocol, the transaction T first issues r_1 requests of t_1 to the replicas of x_1. Then, each instance of t_1 on a replica issues r_2 requests of t_2 to the replicas of x_2. In the protocol N, a method t_2 invoked by each instance of t_1 is performed. Here, totally r_1 times r_2 requests are performed. In the protocol Q, at most one instance of t_2 is performed on each replica of x_2 by the resolution procedure of the redundant invocation. Since the quorum explosion is not resolved, the expected number QE_2 of replicas where t_2 is performed is $n_2[1 - (1 - \frac{r_2}{n_2})^{r_1}]$. Then, each instance of t_2 issues requests of t_3 to r_3 replicas of o_3. Here, $n_3[1 - (1 - \frac{r_3}{n_3})^{r_1 r_2}]$ replicas are locked in the protocol N and $QE_3 = n_3[1 - (1 - \frac{r_3}{n_3})^{QE_2}]$ replicas in the protocol Q. In the protocol R, t_2 is performed on only r_2 replicas of the object x_2.

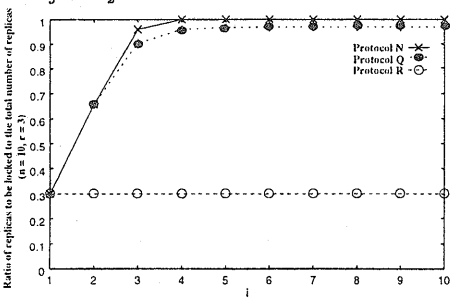

Figure 9: The ratio of replicas to be locked ($n = 10$ and $r = 3$).

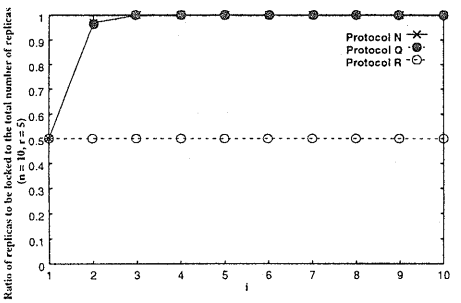

Figure 10: The ratio of replicas to be locked ($n = 10$ and $r = 5$).

In the evaluation, we assume that $n_1 = n_2 = \ldots = n = 10$ and $r_1 = r_2 = \ldots = r$. Figure 9 and 10 show that a ratio of replicas where a method is performed to the number n of the replicas at each invocation level for $r = 3$ and $r = 5$, respectively. The dotted line with white circles shows the protocol R. The straight line indicates the protocol N and the other dotted line with black circles shows the protocol Q. If the methods are invoked at deeper level than two or three, all the replicas are locked.

6 Concluding Remarks

This paper discussed how multiple transactions invoke methods on replicas of objects. The object supports a more abstract level of method. In addition, methods are invoked in a nested manner. It is not required to perform every update method instance on the replica which has been computed on the other replicas if the instance is compatible with the instances performed. We discussed how to resolve redundant invocations and quorum explosions to occur in systems where methods are invoked on multiple replicas in a nested manner. By using the quorum-based locking protocol with the exchanging procedure and resolution of redundant invocations and quorum explosions, an object-based system including replicas of objects can be efficiently realized.

References

[1] Bernstein, P. A., Hadzilacos, V., and Goodman, N., "Concurrency Control and Recovery in Database Systems," *Addison-Wesley*, 1987.

[2] Bernstein, P. A., and Goodman, N., "The Failure and Recovery Problem for Replicated Databases," *Proc. 2nd ACM POCS*, 1983, pp. 114–122.

[3] Carey, J. M. and Livny, M., "Conflict Detection Tradeoffs for Replicated Data," *ACM TODS*, Vol.16, No.4, 1991, pp. 703–746.

[4] Chevalier, P. -Y., "A Replicated Object Server for a Distributed Object-Oriented System," *Proc. of IEEE SRDS*, 1992, pp.4-11.

[5] Garcia-Molina, H. and Barbara, D., "How to Assign Votes in a Distributed System," *JACM*, Vol 32, No.4, 1985, pp. 841-860.

[6] Hasegawa, K. and Takizawa, M., "Optimistic Concurrency Control for Replicated Objects", *Proc. of Int'l Symp. on Communications (ISCOM'97)*, 1997, pp. 149–152.

[7] Hasegawa, K., Higaki, H., and Takizawa, M., "Object Replication Using Version Vector", *Proc. of the 6th IEEE Int'l Conf. on Parallel and Distributed Systems (ICPADS-98)*, 1998, pp. 147–154.

[8] Jing, J., Bukhres, O., and Elmagarmid, A., "Distributed Lock Management for Mobile Transactions," *Proc. of IEEE ICDCS-15*, 1995, pp. 118-125.

[9] Korth, H. F., "Locking Primitives in a Database System," *JACM*, Vol. 30, No. 1, 1983, pp. 55-79.

[10] Silvano, M. and Douglas, C. S., "Constructing Reliable Distributed Communication Systems with CORBA," *IEEE Comm. Magazine*, Vol.35, No.2, 1997, pp.56–60.

[11] Tanaka, K., Hasegawa, K., and Takizawa, M., "Quorum-Based Replication in Object-Based Systems," *Journal of Information Science and Engineering (JISE)*, Vol. 16, 2000, pp. 317–331.