On the Consistency Problem in Mobile Distributed Computing*

Rachid Guerraoui
rachid.guerraoui@epfl.ch

Corine Hari
corine.hari@epfl.ch

Distributed Programming Laboratory
Swiss Federal Institute of Technology (EPFL)
CH-1015 Lausanne

ABSTRACT

This paper describes some preliminary steps towards defining a consistency criteria for mobile replicated systems using operational transformations. Our criterion lies between traditional strong criteria, preventing divergence, and traditional weak criteria, not enforcing any eventual form of convergence. We give a precise definition of our criterion and discuss its use to state the correctness and incorrectness of some existing practical algorithms.

Categories and Subject Descriptors

F.2 [**Theory of Computation**]: Analysis of Algorithms

General Terms

Algorithms, Theory

Keywords

Consistency, Distributed, Operational Transformations

1. INTRODUCTION

1.1 Motivation

Several algorithms have been devised to ensure some form of consistency among replicas in a mobile environment. Many of these algorithms share a common flavor [4, 11, 13, 6, 12, 3]: they seek to allow replica divergence during disconnection and enforce convergence after reconnection. In short, during disconnection periods, operations on the same object might be executed concurrently on replicas of the object and this might typically lead to replicas with different states if the operations conflict (say updates). After reconnection, and provided that the processes hosting the replicas are connected for long enough, it is expected that these replicas eventually converge to the same state.

The problem that these algorithms seek to solve is indeed intuitive but, to our knowledge, it has never been precisely defined for algorithms such as [4] or [13]. This makes it impossible to draw any precise conclusion about their correctness. Having no precise clue about the actual problem also makes any performance comparison potentially unfair between any two algorithms that might be solving different problems.

It is very tempting to explore the possibility of coming up with a precise specification of the problem. This would enable us to understand it better and measure the correctness of the algorithms presumabely solving it. It might also lead to invent new alternative algorithms and possibly state and prove lower bound results on these algorithms.

1.2 Background

Giving a precise specification of the consistency problem in mobile computing comes down to defining a consistency criterion that captures exactly the desired behavior of objects shared by mobile processes.

Basically, traditional consistency criteria can be classified in two classes:

The first class consists of strong consistency criteria which prevent replicas from diverging. These include for instance sequential consistency [7] or linearizability [5]. Roughly speaking, these ensure that any process accessing the shared object has the illusion that operations, even from different processes, are performed on the object one at a time, according to some global order. This is captured by the guarantee that there must be a (imaginary) global serialization of all operations that no process, as far as the results it got from invoking operations on the object, can distinguish from the actual execution of the operations. To ensure such a consistency criterion, no algorithm can enable operations that are conflicting, such as for instance `enqueue` operations on the same `queue` object, to be performed concurrently. This is clearly not adapted to a practical mobile setting where disconnection is the normal case.

The second class consists of weak consistency criteria that allow divergence but might never enforce convergence. A representative of this class is causal consistency [1]. In this case, only operations that are related by a causal precedence are globally ordered. Concurrent operations that are unrelated from a causal perspective do not need to obey

*This work is partially supported by the Swiss National Science Foundation(project no 21-64994.01)

POMC'02, October 30-31, 2002, Toulouse, France.

any global ordering. So even after a reconnection period, and even if the processes hosting the replicas remain permanently connected, every process might end up with its own view of the object state.

In short, for many mobile applications, neither strong consistency nor weak consistency criteria are adapted. One might indeed implement strong consistency in a mobile environment if the shared objects have a very simple type (e.g., read-write semantics), or sometimes live with weak consistency [9]. In general however, some intermediate level of consistency is achieved by algorithms that control the sharing of a document in a mobile environment (e.g., [4, 6, 12, 3]): they indeed enable divergence in disconnected periods but also enforce convergence after reconnection. But what is the actual consistency criterion ensured by these algorithms?

1.3 Approach

A careful look at these algorithms leads to an interesting observation. During the reconnection period, operations previously executed locally on every replica are exchanged and then merged. One can achieve the merging by a simple state transfer primitive as provided by group communication systems [2]: such a primitive is however inherently asymmetric and makes the work performed by all but one process useless. Some algorithms use less drastic schemes to achieve the merging. After identifying the conflicting operations, they might either (a) leave it up to the user of the system to decide, e.g., [3], possibly after attempting to reorder the operations, e.g., [6], or (b) perform some roll-back operations to resolve the conflict [12], or (c) even transform some of the operations into non-conflicting ones, e.g., [4, 11, 13]. In the context of this paper, we are interested in precisely capturing the latter semantics, i.e., (c), where no roll-back or manual intervention is needed, but where operation transformations (called transpositions) must have been planned in advance.

In this case, after a reconnection, a process p might get an operation op previously executed during disconnection by some other process q on a replica of the same object. Process p does not necessarily incorporate op by executing it as is on its replica. Instead, it might execute a variant of op, op' (called a *transposition* [4] of op), that intuitively intends to achieve the same effect as op, if executed after all operations that p had executed concurrently to op. This basically means that the global serialization order that indeed ensures convergence (i.e., that leads to the common convergent state) does not necessarily integrate the original operations executed by the processes, but potentially variants of these operations (the transpositions). This observation was the key to our contribution.

1.4 Contribution

This paper describes some preliminary steps towards defining a consistency criterion for mobile replicated systems using operational transformations. We call this criteria $\diamond$*consistency*. Basically, $\diamond$consistency allows processes to perform (possibly conflicting) operations on different replicas of the same object, even during disconnection periods, i.e., when every process accesses its own replica without communicating with other processes. Nevertheless, $\diamond$consistency enforces all replicas to eventually converge to the same common state provided that the processes reconnect for sufficiently long. This common state would indeed correspond to a state produced by a global serialization. Interestingly, this state is obtained by preserving, in a precise sense, the intentions of all the operations executed during disconnections.

Unlike for a traditional strong consistency criterion however, our global serialization might not incorporate the exact operations executed by the processes having accessed the object, but some variants of these operations: their transpositions. These are typically defined in advance by the application programmer. We present the semantics of these transpositions and accordingly define a new notion of serialization and the resulting consistency criterion: $\diamond$consistency. We use this criterion to discuss the correctness (vs incorrectness) of some practical replication algorithms devised for a mobile computing environment. Maybe even more interestingly, we open some research directions that might help bridge the gap between current practices in mobile environments and more traditional aspects of distributed computing.

2. MODEL

2.1 Shared Objects

We consider the model introduced in [5]. A distributed system consists of a set of processes, each communicating with one another through shared objects. Objects have an associated set of operations, which can modify their state. Operation executions are defined by an invocation and an associated response. When there is no ambiguity, we use the term operation for operation executions. Processes are sequential: no process invokes a second operation without having received the response to the first one.

An execution is represented by a finite sequence of operations, called a *history*. A *process subhistory* $H|p$ of a history H is the subsequence of operations of H executed at a process p. Note that a sequence of operations is simply a set of operations with an order relation.

The *sequential specification* of an object defines the behavior of the object by identifying the set of executions that are valid according to its semantics, when processes access the object in a sequential and failure-free way.

2.2 Consistency Criteria

Roughly speaking, a consistency criterion defines which executions of a distributed system are considered correct.

A consistency criterion is typically defined in terms of *equivalence* to a *legal* sequential history [10]. Two histories are equivalent if they are made of the same operations and one is a permutation of the other. A permutation of a set of operations is a *serialization*.

A serialization is *legal* if it belongs to the sequential specification of the objects involved in the history. Legal serializations of a valid history are for instance obtained by restricting permutations to interchanges of commutative operations [14].

We recall here the definition of a well known *strong* consistency criterion: sequential consistency [7].

- A history H is *sequentially consistent* if there exists a legal serialization S of H such that for each process p, $H|p = S|p$.

With sequential consistency, operations from different processes can be interleaved, but operations from the same process must appear in the same order.

Now we recall the definition of a well known *weak* consistency criterion: causal consistency [1].

Operations can have a causal dependency between them. Let $\xrightarrow{t_1,t_2}$ be the partial order induced by a causal dependency between operations of type t_1 and type t_2[1] for a history H. Let $A^H_{p+t_1}$ be the subhistory composed of $H|p$ and all operations of type t_1 in H.

- A history H is *causal* if *for each process* p there exists a legal serialization γ of $A^H_{p+t_1}$ that respects $\xrightarrow{t_1,t_2}$ and $H|p = \gamma|p$.

In the case of sequential consistency, there exists a single serialization of the history for all processes, whereas with causal consistency, there may be a different serialization for each process. Causal consistency therefore does not guarantee that all processes will have the same convergent state, contrarily to sequential consistency which does not allow any divergence.

3. CURRENT PRACTICES

In this section, we give an intuitive example of an algorithm [11] which ensures the *reconciliation* of diverging replicas in a mobile computing environment. The idea is that operations which were executed on another replica must be transformed to account for operations which were executed concurrently on the current replica. Then we define the notion of transposition which allows for such specific kinds of permutations.

3.1 Example

Figure 1 gives an example of an execution of the algorithm after reconnection. The application modeled is a collaborative text editor, represented by shared String objects with two basic operations: insert and delete. We will use this simple intuitive object type throughout the paper to illustrate the different concepts.

Two processes concurrently execute the operations $op_1 : ins(x, 1)$ and $op_2 : del(3)$, which respectively insert x at position 1 and delete the third character, on their respective replicas. They then broadcast their operations to all processes. If each process executes the operations as it receives them, we will end up in divergent states. If process 1 had *transformed* op_2 to $op'_2 : del(4)$, however, convergence could have been preserved. Intuitively, op'_2 would have included the effect of op_1, which was to insert a preceding character.

3.2 Operational Transformations

The algorithm briefly introduced above relies on the technique of operational transformations [4], which originated in the context of collaborative environments. It consists in transforming operations to execute them at a different point in the history. Intuitively, it transforms an operation to include or exclude the effect of other operations. There are two types of transformations:

- *Forward transposition* (sometimes called *inclusion transformation*) modifies the operation to include the effect of other operations, effectively allowing to move it forward in the history by including the effect of later operations.

[1]The "write-into" relation from [1] is a dependency between write and read operations

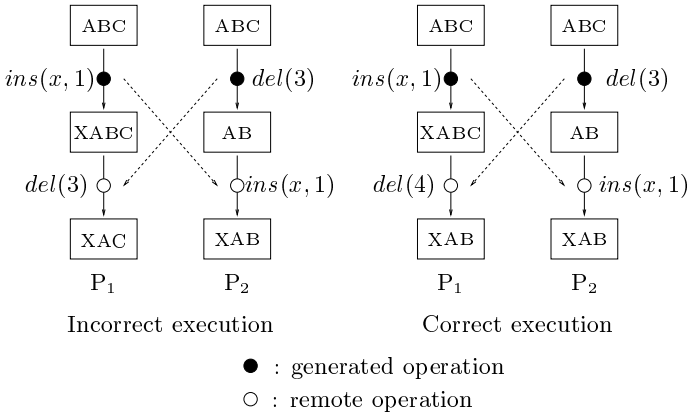

Figure 1: After Reconnection

More precisely: $fwd(op_1, op_2) = op'_2$ such that: $\forall O_i$: Effect$(O_i.op_1, op'_2)$ = Effect(O_i, op_2)

- *Backward transposition* (sometimes called *exclusion transformation*) modifies the operation to exclude the effect of other operations, effectively allowing to move it backward in the history by excluding the effect of earlier operations.
More precisely: $bkwd(op_1, op_2) = op'_2$ such that: $op_2 = fwd(op_1, op'_2)$

The *effect* of an operation is interpreted by the programmer, who must define the transpositions for the shared object type. Note that to "realize the same effect" does not mean "lead to the same state". For instance, the effect of the operations $ins(x, 2)$ on the string bc could be to insert the x between b and c, yielding the string bxc. On a string abc, the operation that realizes the same effect would lead to abxc, obviously different from bxc.

Following our earlier example (Figure 1), consider two operations $op_1 = ins(a, 2)$ and $op_2 = del(3)$ which respectively insert a at the second position and delete the third character. The forward transposition of op_2 with regard to op_1, $fwd(op_1, op_2)$, is $del(4)$. It is the operation which translates the effect of op_2 (deleting the third character) after the execution of op_1 which inserts a preceding character: to delete what used to be the third character, the operation now has to delete the fourth character.

In the same way, the backward transposition of op_2 with op_1, $bkwd(op_1, op_2)$, is $del(2)$. It excludes the effect of op_1 (inserting a preceding character) from the definition of op_2: to delete what would have been the third character if a preceding character had been inserted, the operation now has to delete the second character.

Defining transpositions for the insert and delete operations comes down to testing all different combinations of insertions and deletions of preceding and following characters. The complete definitions of transpositions for text editors can for instance be found in [11] or [13].

Operations are defined on a state. Transpositions, such as they are defined above, modify them to include or exclude the effect of other operations that are defined *on the same state*. Any transposition of operations that are not defined on the same state is invalid[2].

[2]Commutativity [14] is the particular case where transpo-

4. ◇SERIALIZATION

Operation transpositions provide a way to extend the traditional notion of serialization. As seen in Section 2, the precise definition of consistency criteria relies on this notion. In this section we redefine serialization of histories for a setting augmented with operation transformations, which will be used to define a new consistency criterion in this context. We first introduce some definitions and notations to capture various notions around operations and transpositions. Then we define the notion of equivalence between sets of operations. Finally, we define our notion of ◇serialization.

4.1 Notations and Definitions

We introduce here the notations and definitions underlying our new notion of serialization. The complete notation for an operation execution is ${}^{b_i}op_i^{f_i}\langle c_i\rangle$. We denote by c_i the set of operations executed before the generation of op_i and by f_i and b_i the sets of operations that op_i was transposed forward, resp. backward, with. Finally, (op_i) denotes an operation based on op_i.

The context c_i refers to the *defining context*, which is the set of operations that were executed by the process locally before invoking op_i.

If the operations are transposed and re-arranged to produce a new sequence S, then the operations preceding op_i in S are not necessarily the same as the defining context c_i. We call this context the *execution context* of op_i in S, which is noted $exec(op_i)_S$. Note that the defining context of an operation is determined once and for all, whereas the execution context changes.

For instance, consider the operation sequence at process P_1 in Figure 2. Operation op_2's defining context is equal to its execution context, i.e., $c_2 = exec(op_2)_{P_2} = \{op_1\}$. At process P_2, however, op_2's execution context is $\{op1, op_3\}$, whereas its defining context stays the same (as it is defined by its generating process, P_1).

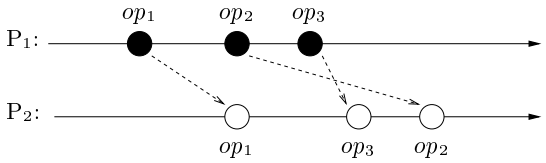

Figure 2: Defining and execution contexts

A *base operation* is one that has not been transposed, i.e., ${}^{\emptyset}op_i^{\emptyset}\langle c_i\rangle$. As c_i does not vary, this notation is often abridged to op_i.

Any operation that is obtained by a transformation of op_i is called a *form* of op_i and is denoted by (op_i). It is obtained from the base operation op_i by transposing it with a sequence of operations. Note that $op_i \in \{(op_i)\}$, i.e., that op_i itself is a form of op_i, as it is simply the particular case where $f_i = b_i = \emptyset$.

4.2 Operation Equivalence

When considering transpositions, operations do not exist solely under a single form anymore. Therefore notions that rely on equivalence of operations have to be redefined.

Two forms of the same operation can produce the same operation. Transposing an operation op_i forward with an operation op_j means that we are including op_j's effect in op_i. This is the same as if op_j figured in op_i's context. The same way, if op_i is transposed backwards with an operation op_k, this excludes op_k's effect from op_i's context. Therefore we can establish the following equivalence relationships between forms of the same operation:

- $op_i\langle op_j\rangle = op_i^{op_j}\langle\emptyset\rangle$
- ${}^{op_k}op_i\langle op_k\rangle = op_i\langle\emptyset\rangle$

From which we deduce that: ${}^{op_l}op_i^{op_l}\langle\emptyset\rangle = op_i\langle\emptyset\rangle$

Moreover, two operations that are two forms of the same base operation must now be considered equivalent in certain cases. This leads us to redefine the notion of equivalence between two sets of operations. Two sets of operations S and S' are said to be *equivalent* iff the three following conditions are satisfied:

1. $|S| = |S'|$
2. $\forall op \in S, \exists (op) \in S'$
3. $\forall op' \in S', \exists op \in S$ s.t. $op' = (op)$

Through our notion of equivalence, we define a bijective mapping between the two sets of operations, where each operation of a set has a corresponding form in the other set.

4.3 ◇Serialization

A serialization is a permutation of operations. Operational transformations provide a way to permute operations in a history which otherwise could not have been permuted, therefore producing more legal serializations for a given history than with a strong consistency criterion like sequential consistency.

Intuitively, we want a ◇serialization to be permutations and tranpositions of operations that preserve the effect of the individual operations and lead to the same final state. To do so, we must restrict the permutations and transpositions in the following manner. We say that a sequence T is a ◇serialization of a sequence S iff the two following conditions are satisfied:

1. The sets of operations of S and T are equivalent (as defined in 4.2).
2. T is obtained by transposing and permuting the operations of S, subject to the following constraint: $\forall op_i \in T$, $exec(op_i)_T$ must equal $c_i \cup f_i \setminus b_i$, with c_i defined by S, and $b_i \subseteq c_i \cup f_i$.

The first condition (1) guarantees that all operations appear in the execution, though not necessarily in their original form.
The second condition (2) translates the fact that an operation must be modified to realize the same effect on a possibly different state. It specifies in what form an operation must exist, depending on its position in the sequence, for the sequence to be a serialization of the other.

sitions are restricted to those that do not actually modify the operation. In other words, op_2 commutes forward with op_1 if $fwd(op_1, op_2) = op_2$. In the same way, op_2 commutes backward with op_1 if $bkwd(op_1, op_2) = op_2$. Note that this does not mean commutativity restricts operations to those that do not modify the *state*: it restricts *transpositions* to those that do not modify the *operations*.

S determines the defining context of the operations. That is, c_i is made up of the operations preceding op_i in S. To be a ◇serialization of S, the operations in T can be transposed and re-arranged, but the permutations must obey certain rules with regard to this given context.

If the execution context of an operation contains operations that do not appear in its defining context, the operation must be transposed forward to include their effect. Similarly, if the defining context of the operation contains operations that are not in the execution context, it must be backward transposed to exclude their effect. In other words, $exec(op_i)_T$ must equal $c_i \cup f_i \setminus b_i$.

Moreover, we must not exclude the effet of operations which were not taken into account in the first place. Therefore we must have: $b_i \subseteq c_i \cup f_i$.

For example, consider the following operation sequences:

- S: $op_1 \,.\, op_2 \,.\, op_3$
- S′: $op_1 \,.\, op_3 \,.\, op_2$
- S″: $op_1 \,.\, {}^{op_2}op_3 \,.\, op_2^{\{^{op_2}op_3\}}$

S′ is not a ◇serialization of S, as $exec(op_3)_{S'} = \{op_1\} \neq \{op_1, op_2\} = c_3 \cup f_3 \setminus b_3$.

For S″ we have: $exec(op_3)_{S''} = \{op_1\} = c_3 \cup f_3 \setminus b_3$ and $exec(op_2)_{S''} = \{op_1, {}^{op_2}op_3\} = c_2 \cup f_2 \setminus b_2$. Therefore S″ is a ◇serialization of S.

5. ◇CONSISTENCY

Intuitively, we want to define a criterion which will allow replicas to diverge, but ensure that they will eventually converge. To guarantee this, the transpositions must satisfy certain conditions, introduced below. We then give a few definitions and state the conditions which the algorithms must satisfy, i.e., the consistency criterion itself.

5.1 Conditions on transpositions

Any transpositions used in this context must inevitably satisfy the following condition:

- C1: $\forall op_i, op_j : op_i.op_j^{op_i} = op_j.op_i^{op_j}$

This is necessary to ensure convergence. This states that the order in which the effects of the operations are realized must not matter, allowing processes to execute operations locally during disconnections and have them integrated later by other processes. The sum of all effects must invariably lead to the same state, expected to be a state compatible with the effects of all the operations. Without this condition, no algorithm could ensure that a system having diverged would eventually converge, as shown in the very simple counter-example in Figure 3.

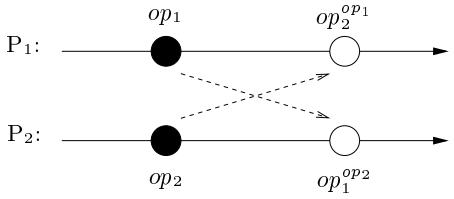

Figure 3: $op_1.op_2^{op_1}$ must equal $op_2.op_1^{op_2}$ for convergence

Likewise, if no other restriction is to be placed on operation order, the following condition must be satisfied.

- C2: $\forall op_i, op_j, op_k : op_k^{\{op_i, op_j^{op_i}\}} = op_k^{\{op_j, op_i^{op_j}\}}$

Consider the case in Figure 3, and suppose both processes now receive a third operation op_3 defined on the same state as op_1 and op_2, i.e., the situation depicted in Figure 4.

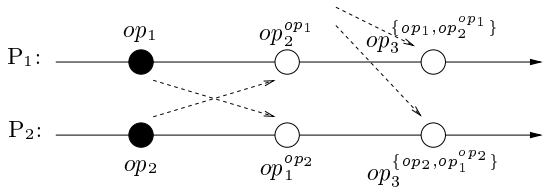

Figure 4: $op_3^{\{op_1, op_2^{op_1}\}}$ must equal $op_3^{\{op_2, op_1^{op_2}\}}$ for convergence

Some algorithms do not require this condition by ensuring a global order on all operations, e.g. SOCT4, which will be presented later.

5.2 Definition

We first introduce the notion of similarity between sequences of operations. Two sequences S and T are similar ($S \sim T$) iff:

- S and T are equivalent (as defined in 4.2).
- The forms of the operations appear in the same order in S as in T.

For example, $op_1.op_2 \sim op_1^{op_x}.op_2^{op_x}$, but $op_1.op_2 \nsim op_2.op_1$. Note that if S and T contain the same forms for each operations, this defines equality between sequences.

We now use our notion of ◇serialization and similarity to define a new consistency criterion: ◇*consistency*.

- A history H is ◇*consistent* iff there exists a legal ◇serialization S of H such that for each process p, $H|p \sim S|p$.
- An algorithm ensures ◇*consistency* iff it produces only ◇consistent histories.

We present here a simple example to illustrate how the different conditions work together to ensure that different ◇serializations of the same base operations lead to the same state. Consider the execution in Figure 5.

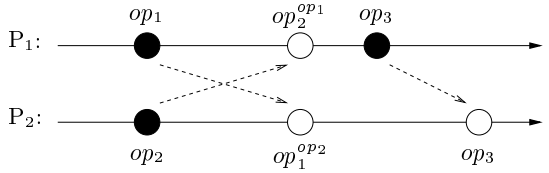

Figure 5: Execution

As the order of the forms of operations in a valid serialization must be the same for each process, and all operations must appear in some form, the possible ◇serializations are A:$(op_1)(op_2)(op_3)$, B:$(op_2)(op_1)(op_3)$, and C:$(op_1)(op_3)(op_2)$.

Condition 2 of ◇serilization specifies what the exact form of the operations must be, when belonging to those particular sequences. This gives us:

- A: $op_1 \ . \ op_2^{op_1} \ . \ op_3$
- B: $op_2 \ . \ op_1^{op_2} \ . \ op_3$
- C: $op_1 \ . \ {}^{op_2^{op_1}}op_3 \ . \ op_2^{\{op_1, {}^{op_2^{op_1}}op_3\}}$

Given that $\forall op_i, op_j : op_i.op_j^{op_i} = op_j.op_i^{op_j}$, sequence B is equal to sequence A, as $op_2.op_1^{op_2} = op_1.op_2^{op_1}$. In the same way, sequence C is equal to $op_1.op_2^{op_1}.^{\{op_2^{op_1}\}}op_3^{\{op_2^{op_1}\}}$, which in turn is equal to A.

5.3 A Correct Algorithm

We now use ◇consistency to state the correctness of an existing algorithm. Namely, we will consider SOCT4 [13] which was designed in the context of collaborative group editors. This algorithm ensures two properties: *convergence* and *causality*.

The notion of causality used in this algorithm states that an operation causally depends on all operations that were previously executed by the generating process. As this is even stronger than ensuring that operations generated by the same process appear in the same order, the second condition for ◇consistency is necessarily ensured.

The algorithm guarantees that all operations will eventually be executed on all processes, though not necessarily in the same form. The first condition for ◇serialization is therefore satisfied and we need only consider the second condition.

SOCT4 ensures that all processes eventually execute the operations in the same global order, consistent with the causal order described above. This means that each process eventually ends up with the ◇serialization we are looking for. During disconnections, however, the individual processes may still execute operations locally. When receiving new operations, they must integrate it with potentially concurrent operations on the local process, as illustrated in Figure 6. This is handled by transposing the new operation forward with all concurrent operations to be able to execute it on the current state, then transposing all the concurrent operations with this new operation to incorporate its effect. This integrates the remote operation in its corresponding place in the global order. Note that the operation to be integrated is executed under a different form ($5'$ in the figure), to be compatible with the current state at the process, but finally integrated in the history in the form it was received (5 in the figure).

As the global order guarantees that all preceding operations are already executed, all operations in the new operation's defining context are already in the execution context. The backward transposition is therefore not needed. There might be concurrent operations already executed on the local process, however, which are in the execution context and not in the defining context. The new operation is transposed forward with them, and therefore the condition that states that the execution context must be equal to $c_i \cup f_i \setminus b_i$ is satisfied.

This algorithm therefore satisfies ◇consistency.

5.4 An Incorrect Algorithm

We now study dOPT, which was the first algorithm designed for operational transformations [4]. It uses only forward transpositions to integrate the remote operations as they are received from other processes. It has previously been shown to produce incorrect executions. We show that the dOPT algorithm is not correct with respect to ◇consistency by exhibiting the execution shown in Figure 7, which does not have a legal ◇serialization.

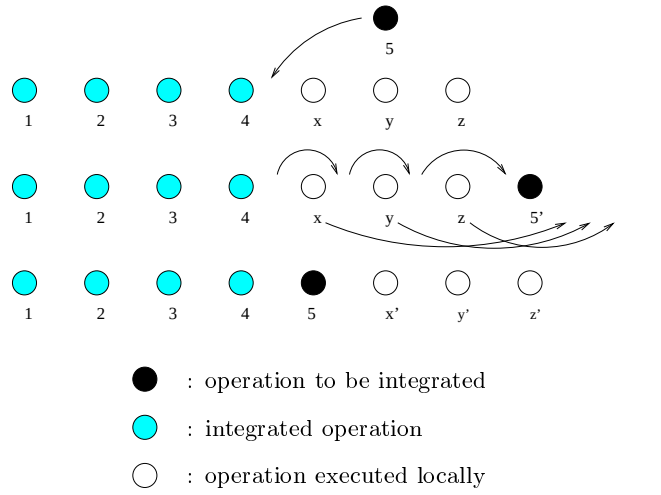

Figure 6: Operation integration in SOCT4

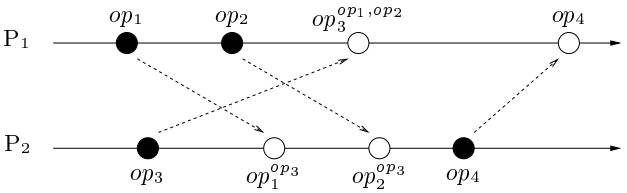

Figure 7: Execution produced by dOPT

Looking at the defining context of the different operations, we see that op_4 has $op_2^{op_3}$ in its context. As op_2 and op_3 are not defined on the same context, this transposition is not valid. There is therefore no way this transposition can belong to op_4's execution context, nor can it be transposed backward with it. No serialization can then satisfy the condition $exec(op_4) = c_4 \cup f_4 \setminus b_4$.

6. CONCLUDING REMARKS

The contribution of this paper is to have taken some preliminary steps towards precisely defining the replica consistency problem in mobile computing. Our motivation was to address the following question: what do practical algorithms that tackle the replication problem in a mobile environment ensure exactly? We propose a partial answer to this question through our new notion of ◇consistency which we tried to define with the same level of rigour as traditional consistency criteria like sequential consistency [7] and causality [1].

Our answer is partial in various senses. We were mainly inspired by algorithms devised in the collaborative world and more precisely algorithms that control the replication of shared documents. These algorithms ensure a desirable eventual convergence property after reconnection while, at the same time and in a rather subtle sense, preserving the initial intention of the processes having accessed the object during disconnections. Other inspirations might lead

to different formulations of ◇consistency and an abstract generalization that would abstract away from our notion of operation of transpositions seems very challenging. An exact characterization of algorithms that ensure ◇consistency, that would allow simple correctness and lower bound proofs, is another challenging issue.

Addressing such questions could, we believe, lead to a fruitful interaction between the rather theoretical world of distributed algorithms and the practitioners of mobile computing. It would also be interesting to implement system support for ensuring such consistency, in various mobiles network architectures (e.g., cellular or ad hoc), along the lines of [8] for instance.

7. REFERENCES

[1] M. Ahamad, J. Burns, P. Hutto, and G. Neiger. Causal memory. In *Proceedings of the International Workshop on Distributed Algorithms (WDAG)*, number 579 in LNCS, pages 9–30. Springer-Verlag, 1991.

[2] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: A communication sub-system for high availability. In *Proceedings of the IEEE International Symposium on Fault-Tolerant Computing (FTCS)*, pages 76–84, July 1992.

[3] P. Cederqvist. *Version management with CVS*. 1992.

[4] C. Ellis and S. Gibbs. Concurrency control in groupware systems. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pages 399–407, Seattle, Washington, USA, May 1989.

[5] M. Herlihy and J. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[6] A. Kermarrec, A. Rowstron, M. Shapiro, and P. Druschel. The Icecube approach to the reconciliation of divergent replicas. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, Aug. 2001.

[7] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, 1979.

[8] R. Prakash and R. Baldoni. Architecture for group communication in mobile systems. In *Proceedings of the IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 235–242, 1998.

[9] R. Prakash, M. Raynal, and M. Singhal. An adaptive causal ordering algorithm suited to mobile computing environments. *IEEE Transactions on Parallel and Distributed Systems*, 41(2):190–204, Mar. 1997.

[10] M. Raynal and A. Schiper. A suite of formal definitions for consistency criteria in distributed shared memories. Technical report, IRISA, 1995.

[11] C. Sun and C. Ellis. Operation transformation in real-time group editors: Issues, algorithms and achievements. In *Proceedings of the ACM Conference on Computer Supported Cooperative Work (CSCW)*, pages 59–68, Seattle, Washington, USA, Nov. 1998.

[12] D. Terry, M. Theimer, K. Peterson, A. Demers, M. Spreitzer, and C. Hauser. Managing update conflicts in Bayou, a weakly connecctd replication storage system. In *Proceedings of the ACM Symposium on Operating System Principles (SOSP)*, Dec. 1995.

[13] N. Vidot, M. Cart, J. Ferrié, and M. Suleiman. Copies convergence in a distributed real-time collaborative environment. In *Proceedings of the ACM Conference on Computer Supported Cooperative Work (CSCW)*, Philadelphia, Pennsylvania, USA, December 2000. ACM Press.

[14] W. Weihl. Commutativity-based concurrency control for abstract data types. *IEEE Transactions on Computers*, 37(12):1488–1505, Dec. 1988.