Computing and Informatics, Vol. 24, 2005, 53–66

APPLYING THE GWO MODEL TO RELAXED COLLABORATIVE SYSTEMS

Constanza Prieto, Yadran Eterovic

Department of Computer Science
Pontificia Universidad Católica de Chile
Casilla 306, Santiago 22, Chile Casilla
e-mail: cprietoy@puc.cl, yadran@ing.puc.cl

Manuscript received 16 December 2004

Abstract. Building collaborative applications is still a challenging task. A collaborative application can be viewed as a class of distributed shared memory system. A distinctive property of these systems is their memory consistency model. In this paper, we argue that there is a relationship between different collaboration styles, on the one hand, and different memory consistency models, on the other. In particular, we propose a practical collaboration style, exemplified by a collaborative electronic organizer, that can be supported by the GWO memory consistency model, a rather relaxed model stricter only than local consistency. The advantage of the proposed style is that it reduces the amount of information that must be exchanged among the processors. Because there have been no propositions of the specific rules—i.e., the protocol—that the processors in a system must follow to implement the GWO model, we also propose a protocol that exactly matches the properties of the model.

Keywords: Memory consistency models; Memory consistency protocols; Collaborative applications; Distributed shared memory systems

1 INTRODUCTION

Computer supported collaborative work involves using computers to help a group of users achieve a goal; in particular, the computers facilitate the communication and sharing of data among the users of a collaborative application [3]. Developing a collaborative application is a challenging task, especially if it has to run on a widely

distributed system or support many users. In these cases, communication costs can be significant in terms of system performance, and therefore special care must be paid to network traffic issues [6]. We face the worst situation in terms of performance when the application requires that all users learn immediately about any changes made to shared data by the other users. On the other hand, at the opposite extreme each user works independently and does not care about the changes made by other users. This situation is not desirable either: communication costs are low and system performance is high, but there is no collaboration among users. We call the degree of collaboration offered by an application to its users a collaboration style.

One way to look at collaborative applications is as distributed shared memory systems. A distributed shared memory (DSM) system is a program that runs on top of a collection of interconnected workstations, communicating only by message passing; that offers programmers the illusion of a shared address space. One key property of DSM systems is the underlying memory consistency model: a specification of the allowable behavior of memory. More precisely, if the input to memory is a set of read and write operations and the output of memory is the collection of values returned by all read operations, the consistency model is a function that maps each input to a set of allowable outputs. The memory implementation guarantees that for any input it will produce some output from this set. The program utilizing the DSM must be written to work correctly for any output allowed by the consistency model [12].

If we look at a collaborative application as a DSM system, then its collaboration style is defined and supported by the underlying memory consistency model. For example, in a chat room it is desirable that all users see the sentences submitted by all other users in the same temporal order [11]; this can be achieved through a sequential model [7]. But in a widely distributed system, using this model is very costly. The more relaxed causal model could be used instead: this model requires that the sentences submitted by each user be seen in the order in which they are submitted and that replies be seen after seeing the sentences to which they reply [5]. If we use the even more relaxed PRAM model, then the only guarantee to users is that the sentences submitted by each user will be seen in the order in which they are submitted; there is no constraint on the perceived relative order of sentences submitted by different users [8].

In this paper we propose the use of a new memory consistency model to support a relaxed collaboration style, and we describe a protocol to implement the model in relaxed collaborative applications. The model is due to Steinke and is called Global Write-read-write Order (GWO) [11]. The relevance of the GWO model is that it is more relaxed than causal consistency, but stricter than pure local consistency (the PRAM model is also more relaxed than causal consistency and stricter than local consistency, but it cannot be compared to the GWO model). In the GWO model, users are only guaranteed to see the replies to a sentence after the sentence itself; therefore, each user's sentences are not necessarily seen in the order in which they were submitted.

In a related work, Roberts and Sharkey [9] describe the use of a sufficient causal ordering for an arena-like distributed virtual reality system named PaRADE [9]. They verified through implementation experiences the performance benefits of using a relaxed mechanism to provide memory consistency over a network. The mechanism is based on using time stamps for causally relating events in the system. Our approach is different in that it does not use timestamps. In another work, Shen and Sun [10] describe a flexible notification mechanism for collaborative applications. The mechanism is used to exchange messages, to provide group awareness and/or to maintain consistency of shared artifacts. They also propose a framework to describe and compare a range of notification strategies used in existing collaborative applications.

The rest of the paper is organized as follows. In Section 2, we sketch the behavior of a collaborative application as it is supported by different memory consistency models. In Section 3, we provide the protocol that implements the GWO model. In Section 4, we describe in detail the behavior of a collaborative application supported by the GWO protocol; and in Section 5, we present some conclusions and suggest future work.

2 COLLABORATION UNDER DIFFERENT MEMORY CONSISTENCY MODELS

A memory consistency model for a DSM system is a contract between the software and the memory, establishing that if the software follows certain rules, then the memory will work in a certain manner considered correct [1]. For a collaborative application, the underlying model establishes which consistency checks the users are willing to give up, for example, in exchange for better system performance. In general, the stricter the consistency model the worse the system performance, and vice versa. And essentially, under less strict models users are not guaranteed to all have the same view of the memory at all times during system execution. Figure 1 illustrates the relative strictness of several known memory consistency models: the higher up in the figure, the stricter the model. Strict consistency, at the top, is the strictest model, while local consistency, at the bottom, is the most relaxed.

As we discussed in the introduction, different collaboration styles require memory consistency models of different degrees of strictness. Therefore, each collaboration style requires the exchange of different amounts of information, and with different periodicity. Table 1 exemplifies the collaboration styles supported by different memory consistency models, by describing the behavior of a collaborative canvas application.

An important concept to understand the canvas behavior under both causal consistency and GWO consistency in Table 1 is potential causality. Assume user U writes a value to shared variable x; then, user V reads x's value and writes a value to shared variable y. The write operations on x by U and on y by V are potentially

causally related, because the value written to y may depend on the value read from x. On the other hand, if two users write values to two different variables simultaneously and spontaneously, these operations are not causally related; they are called concurrent.

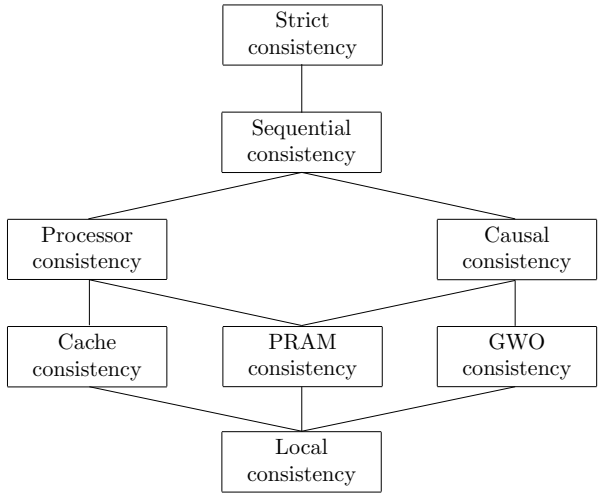

Fig. 1. Relative strictness of known memory consistency models: The strict model, at the top, is the strictest; the local model, at the bottom, is the least strict

3 AN IMPLEMENTATION PROTOCOL FOR THE GWO MODEL

Suppose we have the following collaboration style:

- Each user works most of the time individually on her own data area.
- Each user, at some point in time, needs to know some updates made by other users to the data in her own data area.
- No disjoint collaborative data areas are formed among users.

We propose using the GWO memory consistency model as the underlying model for this collaboration style. The GWO model rule is the following: potentially causally related write operations to variables must be seen in the same order by all users. This means that concurrent write operations—which are not potentially causally related—may be seen in different orders by different users, and that write operations made by one user may be seen by other users in an order different than the order in which the operations were made.

Memory consistency model	**Collaborative canvas behavior**
Strict consistency	All users see all changes made to the canvas as soon as the changes are made.
Sequential consistency [7]	All users agree on the global time ordering of all the changes made to the canvas, but they do not necessarily see the changes as soon as the changes are made; local changes made by each user appear on this global ordering in the order in which they are made.
Processor consistency [4]	Changes made to the canvas by a single user are seen by all other users in the order in which they are made; changes made by different users can be seen in different relative order. Changes made to a single pixel are seen in the same order by all users; changes made to different pixels can be seen in different relative order by different users.
Causal consistency [5]	All users see potentially causally related changes in the same order; besides, all users see the changes made by each user in the order in which they are made.
Cache consistency [4]	All changes to a specific canvas pixel are executed in a sequential order that respects each user's order; all users agree on the order in which operations on a particular pixel are made, and they may see changes to different pixels in different relative order.
PRAM consistency [8]	Changes made by a single user are seen by all other users in the order in which they are made; different users can see the changes made by different users in different relative order.
GWO consistency [11]	All users see potentially causally related changes in the same order; they see any other changes in any order.
Local consistency [2]	Each user sees his/her own changes in the order in which they are made; there is no constraint on the order in which changes made by other users have to be seen.

Table 1. Meaning of each memory consistency model for users of a collaborative canvas

3.1 Protocol Overview

In general terms, our protocol for implementing the GWO model behaves as follows:

- Each processor has a list of messages to be sent and a list of messages received. The messages are transmitted by copying them.
- Each time processor P writes a value to a shared variable x, and it has not performed any read operation before, it produces and stores a message in its send list. The message is not sent to other processors in the system, until another processor wants to read the value written in x by P.

- Each time processor P writes a value to a shared variable x, and it has performed a read operation before, it updates the messages stored in its receive list according to the write operation being executed. The messages are moved from the receive list to the send list, to inform the other processors about the order of the potentially causally related write operations.
- Each time processor P reads the value of a shared variable x, it can read its local value or it can read the value from any processor Q. In the first case, P makes a copy of the corresponding message from its own send list and stores this copy in its receive list. In the second case, P marks in Q one of the messages related to x, stores a copy of this message in its receive list and stores the received value as its local value. The message copied from Q may be any of the (possibly) several messages regarding x, not necessarily the last one generated.

We distinguish two kinds of write operations:

- Potentially causally related write operations. These will be seen in the right order by all processors in the system.
- Concurrent write operations. These may be seen by each processor in a different order.

3.2 Protocol Specification: Notation

We consider a system with n processors, P, Q, ..., and a set of shared variables x, y, ... Each processor P has a local value for each shared variable x; this value is stored in a repository x_p.

Our protocol is based on the exchange of messages by the processors. A message is stored locally by the processor that generated it; it may be sent to another processor; and it may be marked by any of the other processors. There are two kinds of messages:

- Simple messages: Each message generated by processor P is represented by a $n+2$ tuple $(x, v, i_P, M_Q, M_R, \ldots)$, where x is a shared variable, v is the variable's value in P, i_P is P's identification number, and $M_Q, M_R, \ldots$ are $n-1$ fields used by each of the other processors to mark the message. Initially, these fields are set to $\emptyset$. For example, a message from processor P regarding variable x with value v in a system with 3 processors initially has the form $(x, v, i_P, \emptyset, \emptyset)$.
- Double messages: These are composed of two simple messages plus an order relation, $<$, representing the potentially causality relation between the two simple messages. Thus, double messages look like $(x, v, i_P, M_Q, M_R, \ldots) < (y, w, i_Q, M_P, M_R, \ldots)$, where the second message is posterior to the first one.

Each processor P manages a list of messages to be sent to other processors, called P's message pool and denoted by $Pool(P)$; initially, this looks like $[(x, v, i_P, \emptyset, \ldots, \emptyset), (y, z, i_P, \emptyset, \ldots, \emptyset), \ldots]$. The messages in the pool are not sorted and they can be sent

in any order. This is the main difference between this protocol and any protocol that implements causal consistency. Our protocol relaxes the constraint that the write operations of each processor must be seen respecting the processor's local order in which they were executed.

Each processor P also manages a reception list, $Rcv(P)$, to store the messages received from other processors or from itself. The reception list can be represented as a set of messages $Rcv(P) = [(x, v, i_Q, \emptyset, \ldots, \emptyset), (y, w, i_R, \emptyset, \ldots, \emptyset), \ldots]$. The messages in $Rcv(P)$ are not sorted either.

3.3 Protocol Execution: An Example

Consider a system with two processors, P and Q, and two shared variables x and y; the processors execute and interact as seen in Figure 2:

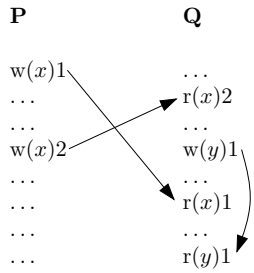

Fig. 2. An example execution of the GWO protocol: w$(x)v$ represents a write operation of the value v to shared variable x; r$(y)u$ represents a read operation on the shared variable y obtaining value u

1. P writes the value 1 in x, stores this value in its repository x_P, generates the message $(x, 1, i_P, \emptyset)$ and stores this message in $Pool(P)$.
2. P writes the value 2 in x, stores this value in its repository x_P, generates the message $(x, 2, i_P, \emptyset)$, and adds this message to $Pool(P)$, that now looks like $[(x, 1, i_P, \emptyset), (x, 2, i_P, \emptyset)]$.
3. Q reads variable x from P; in particular it reads the value 2 by looking at message $(x, 2, i_P, \emptyset)$ in $Pool(P)$. Q updates this message in $Pool(P)$ by marking it with its processor's identification number; the message now looks like $(x, 2, i_P, i_Q)$. Q also stores a copy of this updated message in its reception list $Rcv(Q) = [(x, 2, i_P, 2)]$. Finally, Q stores the value 2 in its repository x_Q.
4. Q writes the value 1 in y, stores this value in its repository y_Q and generates the message $(y, 1, i_Q, \emptyset)$. It then removes all the messages in $Rcv(Q)$ and relates these messages to the message just created, storing the result in $Pool(Q) = [(x, 2, i_P, 2) < (y, 1, i_Q, \emptyset)]$. The original message $(y, 1, i_Q, \emptyset)$ is not stored as a simple message.

5. Q reads again variable x from P; in particular, it reads a value from $Pool(P)$, but this time it reads the value 1 by looking at the message $(x, 1, i_P, \emptyset)$. Q then marks the message with its identification number, $(x, 1, i_P, i_Q)$, stores a copy of this message in its reception list $Rcv(Q) = [(x, 1, i_P, i_Q)]$, and stores the value 1 in its repository x_Q.
6. Finally, Q reads variable y from its repository y_Q, takes from its $Pool(Q)$ a copy of the tail of the message $[(x, 2, i_P, 2) < (y, 1, y_Q, \emptyset)]$ (generated in step 4), and adds it to its list $Rcv(Q)$, resulting in $Rcv(Q) = [(x, 1, i_P, 2), (y, 1, i_Q, \emptyset)]$.

3.4 Protocol Specification: Rules

The protocol is defined by five rules (rules 0 to 4). These rules apply according to the operations that take place in the system. Rules 0 and 1 are set-up rules: Rule 0 establishes that each processor executes its operations in the usual sequential order; and rule 1 specifies the initialization of the system. Rules 2, 3 and 4 are operational rules: Rule 2 specifies the actions that must be performed when a write operation is executed; rule 3 specifies the actions that must be performed when a read operation is executed; and rule 4 specifies the steps to follow for the communication between processors.

Rule 0. All operations executed by the same processor are executed in the order specified by the processor's program: if operation op_1 precedes operation op_2 in the processor's program, then op_2 cannot start its execution if op_1 has not yet finished its execution according to the processor.

Rule 1. Initially, all shared variables are set to the value ε; thus, initially, all repositories in all processors store only this value: $x_P = \varepsilon$, for every processor P and every shared variable x. Also, every message pool and reception list are initially empty: $Pool(P) = \emptyset$ and $Rcv(P) = \emptyset$, for every processor P.

Rule 2. When processor P writes the value v to shared variable x, the following three actions take place:

a) P stores in its repository for x the value v, which becomes the valid value for x in P; $x_P = v$.
b) P generates a simple message about this operation: $(x, v, i_P, \emptyset, \ldots, \emptyset)$; see Section 3.1.
c) P checks to see if its list of received messages $Rcv(P)$ contains any messages. If $Rcv(P)$ is empty, then P simply appends to $Pool(P)$ the message $(x, v, i_P, \emptyset, \ldots, \emptyset)$. Otherwise, each simple message $(y, w, *, \ldots, *)$ is removed from $Rcv(P)$. P creates a double message, by appending the message just produced (step b) to the message removed, and appends it to its message pool, $Pool(P)$, which has the form $(y, w, *, \ldots, *) < (x, v, i_P, \emptyset, \ldots, \emptyset)$. Each double message $m_1 < m_2$ is moved from $Rcv(P)$ to $Pool(P)$. In this case,

P creates a new double message by replicating the tail m_2 and appending to m_2 the message $(x, v, i_P, \emptyset, \ldots, \emptyset)$. Both double messages are appended to $Pool(P)$: first, the copy of the message $m_1 < m_2$, and then, the new message. Thus, the messages appended to $Pool(P)$ have the form $m_1 < m_2, m_2 < (x, v, i_P, \emptyset, \ldots, \emptyset)$.

Rule 3. When processor P reads a shared variable x, it stores x's value v in x_P; this value has one of two possible origins:

a) The value v was written by processor P itself (by applying rule 2 above); then P takes the following actions: First, P searches $Pool(P)$ for the unique message that involves the variable x and the value v. The message can be simple, $(x, v, i_P, *, \ldots, *)$, or double, $m < (x, v, i_P, *, \ldots, *)$. Then, P copies the message found to the receive list $Rcv(i)$. If the message was found within a double message, only the portion of the message that contains the value v is copied. The message copied is $(x, v, i, *, \ldots, *)$.

b) The value v is taken from a processor QP, in particular, from a message $(x, v, i_Q, *, \ldots, *)$ taken from $Pool(Q)$. This operation stores the value v in the repository x_Q. The message is obtained following rule 4.

Rule 4. When processor P receives a message from processor Q's $Pool(Q)$, $P \neq Q$, the following two possibilities have to be considered:

a) P receives a simple message from $Pool(Q)$. In this case, P searches $Pool(Q)$ for messages of the form $(x, v, i_Q, *, \ldots, *)$, i.e., involving the variable x and not marked by P. If P finds no such messages, it performs the read operation explained above. Otherwise, P selects any of the messages, updates the selected message in $Pool(Q)$ by appending its identification number, i_P, to it, copies the message to its receive list $Rcv(P) = [*, (x, v, i_Q, i_P, *, \ldots, *)]$, and updates its local repository with the new value v for x; $x_P = v$.

b) P receives a double message from $Pool(Q)$. In this case, actions are similar to Case 1, but take into account the fact that they are dealing now with double messages. P searches $Pool(Q)$ for messages with tails involving variable x and not marked by P, i.e., messages of the form $(*, *, *, *, \ldots, *) < (x, v, i_Q, *, \ldots, *)$. If P finds no such messages, then it performs the read operation explained above. Otherwise, P selects any of the messages, updates the selected message in $Pool(Q)$ by appending its identification number, i_P, to it in the portion regarding variable x (in other words, in the message's tail $(*, *, *, *, \ldots, *) < (x, v, i_Q, *, \ldots, *)$, copies the double message to its receive list $Rcv(P)$, and updates its repository with the new value v for x taken from the message just marked; $x_P = v$.

	A	*B*	*C*
9:00 am	See the doctor		
10:00		Meeting with customer	
11:00			Take car to workshop
1:30 pm			Lunch outside the office
4:00		Meeting with operations	
5:00			Present commercial plan
7:30	Parents meeting at school		
10:00		Mom's birthday	

Table 2. Individual appointments entered to a collaborative electronic organizer: An example

4 BEHAVIOR OF A COLLABORATIVE APPLICATION USING THE GWO PROTOCOL

To illustrate the usage of the GWO protocol in collaborative applications, consider a collaborative electronic organizer that allows users to schedule appointments in their own local copies and also to schedule group meetings. Each user schedules the appointments not involving other users in the system in her local copy. When user A wants to meet with one or more of the other users at a particular time slot, s/he checks the availability of each user for that slot by querying her own schedule. If all users are available, A books the time slot in the local copy of each of the other users, thus actually scheduling the meeting.

However, if user B has already booked the time slot, then A has to look for another time slot to schedule the meeting. This resulting meeting (if any) is potentially causally related with the meeting already in place in B's schedule. The local copies of the schedules of all the users involved record that the new meeting is scheduled at the new time slot because there was a meeting already scheduled for the originally proposed time slot by user B.

The purpose of using GWO in this case is to allow each user to schedule his/her appointments without having to tell everybody else about them, thus informing other users only if and when they request the information. Users who do not want to schedule meetings with a particular user do not need to know about that user's appointments. User's appointments that are not potentially causally related to other appointments do not need to be known by other users at all, thus reducing network traffic.

For example, suppose users A, B and C have scheduled their appointments for today as shown in Table 2. Now suppose user A wants to schedule a meeting with

Collaborative organizer	**GWO protocol**	**Description**
Action: System initialization	Operation: Protocol initialization	System initialization with an "empty" organizer
Item: Each time slot in each user's local view	Variable: Each processor's local variables	Each user has his/her own time slots to schedule personal appointments
Item: Each time slot of organizer	Variable: Shared variables among all processors; each processor has its own view of the organizer	The organizer displays the time slots booked by the users
Action: Schedule a private appointment at time slot x	Operation: Write a value to local variable x and record this information	The user schedules her own appointment and triggers a message reporting that the corresponding time slot is no longer available; the message will only be sent upon request from other users
Action: Delete private appointment at time slot x	Operation: Overwrite initialization value to local variable x and record this information	Deleting an appointment is equivalent to overwriting the initialization value to the variable representing the corresponding time slot
Action: Attempt to schedule meeting with user j at time slot x	Operation: Read the value of local variable x of user j	The organizer corresponding to the user j sends the appointment scheduled in slot x (recorded information) to the requesting user
Item: Online view of each user's organizer	Variable: Each processor's set of received messages, with information about other processor's unavailable time slots	Messages from user i are received according to an user's interest in scheduling meetings with user i
Action: View final state of the organizer	Operation: All messages are sent to all processors in the system, or they are discarded	Final state at the end of the day, when all unsent messages are sent to all processors (or they are simply deleted)
Action: Learn about potentially causally related meetings	Operation: All potentially causally related messages are sent to all processors	Each user sees, in his/her local view of the organizer, potentially causally related meetings in the order in which they were scheduled

Table 3. Equivalence between the collaborative electronic organizer's items and actions, and GWO protocol's variables and operations

user B at 4 pm. When A checks B's appointments, she realizes that B already has an appointment at 4 pm, so she schedules the meeting at 5 pm: B's 4 pm appointment becomes potentially causally related to A and B's 5 pm meeting. Thus, if later on C learns about A and B's 5 pm meeting, he must also learn (and must learn first) about B's 4 pm appointment. Causality in this case is due to the fact that a particular meeting time was set taking into account a previously scheduled meeting.

5 CONCLUSIONS

In the realm of collaborative applications, we have shown that there is a mapping between different collaboration styles, on the one hand, and different memory consistency models, on the other. Specifically, collaboration styles that require that all users see the same data at the same time must be supported by strict memory consistency models, while relaxed memory consistency models can be used to support collaboration styles in which users only exchange information on a need to know basis. Therefore, neither all collaboration styles require the exchange of the same large amount of information, nor do they have to exchange the information at the same times. This fact can be used to improve the performance of certain collaborative applications.

We have also described a particular collaboration style – exemplified by a collaborative electronic organizer – which we mapped to a very relaxed memory consistency model: GWO. Because in GWO information is exchanged only between pairs of users and only when a user requests it, the existence of this mapping means that it is possible to collaborate effectively without having to exchange large amounts of information all the time.

Finally, we have introduced a protocol that implements GWO and that can in practice be used to support the collaboration style illustrated by the electronic organizer. We have described the protocol in detail, and we provided a summary of the meaning of the most relevant collaborative actions in terms of the protocol operations.

As future work, we are planning to implement the GWO protocol in a distributed system to support a collaborative environment and test its impact on performance and network load.

REFERENCES

[1] Adve, S.—Hill, M.: Weak Ordering – A New Definition. Proc. 17 Annual Intl. Symp. on Computer Architecture, 1990, pp. 2–14.

[2] Bataller, J.—Bernabeu, J.: Synchronized DSM Models. In C. Lengauer, M. Griebl, and S. Gorlatch, editors, Proc. Third Intl. Euro-Par Conf., Berlin, 1997, pp. 468–475.

[3] Ellis, A.—Gibbs, J.—Rein, L.: Groupware, Some Issues and Experiences. Communications of the ACM, Vol. 34, 1991, No. 1, pp. 38–58.

[4] Goodman, J.: Cache Consistency and Sequential Consistency. Technical Report 61. IEEE Scalable Coherent Interface Working Group, 1989.

[5] Hutto, P.—Ahamad, M.: Causal Memory. Lecture Notes on Computer Science. Vol. 579 (Proc. Fifth International Workshop on Distributed Algorithms), Springer-Verlag, 1991, pp. 9–30.

[6] Kuroda, M.—Ono, R.—Shimotsuma, Y.—Watanabe, T.—Mizuno, T.: Data Transfer Evaluation of Nomadic Data Consistency Model for Large Scale Mobile Systems. IEICE Transactions on Information and Systems, 1999, E82-D(4): 822.

[7] Lamport, L.: How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. IEEE Transactions on Computers. Vol. 28, 1979, No. 9, pp. 690–691.

[8] Lipton, R.—Sandberg, J.: PRAM: A scalable shared memory. Technical Report 180–88, Department of Computer Science, Princeton University, 1988.

[9] Roberts, D.—Sharkey, P.: Maximising Concurrency and Scalability in a Consistent, Causal, Distributed Virtual Reality System, Whilst Minimising the Effect of Network Delays. Proc. 6th Intl. Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises (WET-ICE), IEEE Computer Society, 1997, pp. 161–166.

[10] Shen, H.—Sun, C.: Flexible Notification for Collaborative Systems. Proc. ACM Conference on Computer Supported Cooperative Work, CSCW 2002. New Orleans, 2002, pp. 77–86.

[11] Steinke, R.: Consistency Model Transitions in Shared Memory. Ph.D. Thesis, Department of Computer Science, University of Colorado, 2001.

[12] Steinke, R.—Nutt, G.: A Unified Theory of Shared Memory Consistency. Journal of the ACM, Vol. 51, No. 5, Sept. 2004, pp. 800–849.

Constanza Prieto is Quality Manager at Netred, a software development organization in Santiago, Chile. She focused her research on protocols for memory consistency over distributed shared memory systems.

Yadran Eterovic teaches and does research at the Department of Computer Science, Pontificia Universidad Catolica de Chile. His research concentrates on software design, including concurrent/distributed programming,and object-oriented and aspect-oriented design.