Total Order Communications: a Practical Analysis*

Roberto Baldoni, Stefano Cimmino, and Carlo Marchetti

Dipartimento di Informatica e Sistemistica
Università di Roma "La Sapienza"
Via Salaria 113, 00198, Roma, Italy
{baldoni,cimmino,marchet}@dis.uniroma1.it

Abstract. Total Order (TO) broadcast is a widely used communication abstraction that has been deeply investigated during the last decade. As such, the amount of relevant works may leave practitioners wondering how to select the TO implementation that best fits the requirements of their applications. Different implementations are indeed available, each providing distinct safety guarantees and performance. These aspects must be considered together in order to build a correct and sufficiently performing application. To this end, this paper analyzes six TO implementations embedded in three freely-distributed group communication systems, namely Ensemble, Spread and JavaGroups. Implementations are first classified according to the enforced specifications, which is given using a framework for specification tailored to total order communications. Then, implementations are compared under the performance viewpoint in a simple yet meaningful deployment scenario. In our opinion, this structured information should assist practitioners (i) in deeply understanding the ways in which implementations may differ (specifications, performance) and (ii) in quickly relating a set of total order algorithms to their specifications, implementations and performance.

1 Introduction

Total Order (TO) is a widely investigated communication abstraction implemented in several distributed systems. Intuitively, a TO primitive ensures that processes of a message-passing distributed system deliver the same sequence of messages. This property is extremely useful for implementing several applications, e.g active software replication [1].

However, there are several subtleties that still deserve clarification, especially among practitioners that can get confused by the relevant amount of work done in this area. A first issue is to understand the guarantees of a TO primitive, as distinct primitives and implementations enforce distinct specifications that have to be matched against application correctness requirements. To achieve

* This work is partially supported by the european project EUPubli.com funded by the European Community and by the italian project MAIS funded by the Italian Ministry of Research

this, in this paper we first present six existing TO specifications organized into a hierarchy, and then we identify how specifications differ in terms of the possible behavior of faulty processes. Then, we classify into the hierarchy both fixed sequencer and privilege-based TO protocols given in the context of primary component group communications [2, 3], by also pointing out real systems implementing these primitives. These are the results of a formal analysis available in a companion paper [4].

A further issue we deem relevant for practitioners is performance. Several works present performance analysis of TO primitives, e.g. [5, 6]. Some other works discuss the correlation between the guarantees and the achievable performance of a TO implementation, e.g. [7]. These works mainly focus on intrinsic characteristics of the analyzed primitives, and not on the overall system in which a primitive is typically implemented. Therefore in this paper, in order to assist practitioners in finding the TO implementation that best matches both applications' correctness and performance requirements, we present a simple yet meaningful performance analysis of the implementations in real systems of the discussed TO primitives.The results show that the performance of a TO primitive depends on the combination of three factors, (i) the enforced TO specification, (ii) the TO protocol used to implement that specification, and (iii) the way the protocol is implemented.

The remainder of this paper is organized as follows. Section 2 introduces total order broadcast. In particular, it describes the system model, the properties defining the TO problem, a hierarchy of TO specifications, and highlights their differences in terms of the admitted behavior of faulty processes. Then, Section 3 presents fixed-sequencer and privilege-based TO implementations provided by group communication systems. Section 4 describes some real systems implementing TO primitives and compares them from a performance point of view (Appendix A gives further details about the configuration of these systems). Finally, Section 5 concludes the paper.

2 Total Order Broadcast

2.1 System model

Asynchronous distributed system. We consider a system composed by a finite set of processes $\Pi = \{p_1 \dots p_n\}$ communicating by message passing. Each process behaves according to its specification until it possibly crashes. A process that never crashes is *correct*, while a process that eventually crashes is *faulty*. The system is asynchronous, i.e. there is no bound known or unknown on message transfer delays and on processes' relative speeds. In order to broadcast a message m, a process invokes the $\mathsf{TOcast}(m)$ primitive. Upon receiving a message m, the underlying layer of a process invokes the $\mathsf{TOdeliver}(m)$ primitive, which is an upcall used to deliver m to the process. We say that a process $p \in \Pi$ *tocasts* a message m *iff* it executes $\mathsf{TOcast}(m)$. Analogously, we say that a process $p \in \Pi$ todelivers a message m *iff* it executes $\mathsf{TOdeliver}(m)$.

Properties and specifications. Each process $p \in \Pi$ can experience the occurrence of three kinds of events, namely $TOcast(m)$, $TOdeliver(m)$ and $crash$. An history h_p is the sequence of events occurred at p during its lifetime. A *system run* is a set of histories h_{p_i}, one for each process $p_i \in \Pi$. Informally speaking, a *property* P is a predicate defining a set R_P of system runs, composed by all system runs whose process histories satisfy P. A specification, denoted $S(P_1 \dots P_m)$ (with $m \geq 1$) is the conjunction of m properties, thus defining a set R_S of system runs, composed by those runs satisfying all properties in S. Given two specifications $S(P_1 \dots P_m)$ and $S'(P'_1 \dots P'_\ell)$, we say that S is stronger than S', denoted $S \rightarrow S'$, *iff* $R_S \subset R_{S'}$. In this case we also say that S' is weaker than S. Finally, two specifications S and S' are said to be equivalent, denoted $S \equiv S'$, *iff* $R_S \equiv R_{S'}$.

2.2 Total order properties

Total order broadcast is specified by means of four properties, namely *Validity*, *Integrity*, *Agreement*, and *Order*. Informally speaking, a *Validity* property guarantees that messages sent by correct processes are eventually delivered at least by correct processes; an *Integrity* property guarantees that no spurious or duplicate messages are delivered; an *Agreement* property ensures that (at least correct) processes deliver the same set of messages; an *Order* property constrains (at least correct) processes delivering the same messages to deliver them in the same order. Each property can be formally defined in distinct ways, thus generating distinct specifications. As an example, properties can be defined as *uniform* or *non-uniform*, being non-uniform ones less restrictive, as they allow arbitrary behavior for faulty processes.[1]

Order properties can be further distinguished on the basis of the possibility to have gaps in the sequence of messages delivered by processes, and are thus classified into *strong* and *weak* properties. A weak *Order* property requires a pair of processes delivering the same pair of messages to deliver them in the same order. This restriction does not prevent a process p to skip the delivery of some messages. Therefore, it allows the occurrence of gaps in the sequence of messages delivered by p with respect to those delivered by other processes. In contrast, a strong *Order* property avoids gaps in the sequence of delivered messages as it requires that two processes delivering a message m have delivered exactly the same ordered sequence of messages before delivering m.

Table 1 reports the definition of each property. In particular, we consider both uniform and non-uniform formulations for *Agreement*, i.e. *Uniform Agreement* (UA) and *Non-uniform Agreement* (NUA), and the four *Order* properties arising from the combination of uniform and non-uniform with strong and weak formulations, i.e. Strong Uniform Total Order ($SUTO$), Strong Non-uniform Total Order ($SNUTO$), Weak Uniform Total Order ($WUTO$) and Weak Non-uniform

[1] It is worth noting that uniform properties are meaningful only in certain environments. For instance, uniform properties are not enforceable assuming malicious fault models.

		VALIDITY AND INTEGRITY PROPERTIES
NUV	$\triangleq$	If a *correct* process tocasts a message m, then it eventually todelivers m
UI	$\triangleq$	For any message m, every process p todelivers m at most once, and only if m was previously tocast by some process
		AGREEMENT PROPERTIES
UA	$\triangleq$	If a process todelivers a message m, then all correct processes eventually todeliver m
NUA	$\triangleq$	If a *correct* process todelivers a message m, then all correct processes eventually todeliver m
		ORDER PROPERTIES
$SUTO$	$\triangleq$	If some process todelivers message m before message m', then a process todelivers m' only after it has todelivered m
$SNUTO$	$\triangleq$	If some *correct* process todelivers message m before message m', then a *correct* process todelivers m' only after it has todelivered m
$WUTO$	$\triangleq$	If processes p and q both deliver messages m and m', then p delivers m before m' if and only if q delivers m before m'
$WNUTO$	$\triangleq$	If *correct* processes p and q both todeliver messages m and m', then p todelivers m before m' if and only if q todelivers m before m'

Table 1. Definition of the properties defining TO specifications

Total Order ($WNUTO$). Finally, we consider *Non-uniform Validity* (NUV) and *Uniform Integrity* (UI), as the latter can be easily implemented, thus appearing in almost all TO specifications, while the former is the only *Validity* property meaningful in our system model (i.e. *Uniform Validity* cannot be implemented). Interested readers can refer to [4] for deeper explanations of differences and relations among these properties.

2.3 A hierarchy of total order specifications

Assuming NUV and UI, it is possible to combine *Agreement* and *Order* properties to obtain six significant TO specifications. We denote $TO(A, O)$ the TO specification $S(NUV, UI, A, O)$, where $A \in \{UA, NUA\}$ and $O \in \{SUTO, WUTO, WNUTO\}$.[2]

It is possible to identify several $\rightarrow$ relations among these TO specifications [4]. Figure 1 shows that these specifications represent a hierarchy by depicting the transitive reduction of the $\rightarrow$ relation among TO specifications.

Let us note that the root of the hierarchy, i.e. $TO(UA, SUTO)$, is the specification closest to the intuitive notion of total order broadcast, as it imposes that *the set of messages delivered by each process is a prefix of the ordered set of messages that is delivered by all correct processes*. In contrast, weaker specifications admit runs in which faulty processes may exhibit a larger set of behaviors, as discussed in the following section. It is worth noting that weaker specifications are implemented in several real systems, e.g. Ensemble [8], JavaGroups [9].

[2] In [4] we show that $TO(NUA, SNUTO) \equiv TO(NUA, WNUTO)$ and that $TO(UA, SNUTO) \equiv TO(UA, WNUTO)$.

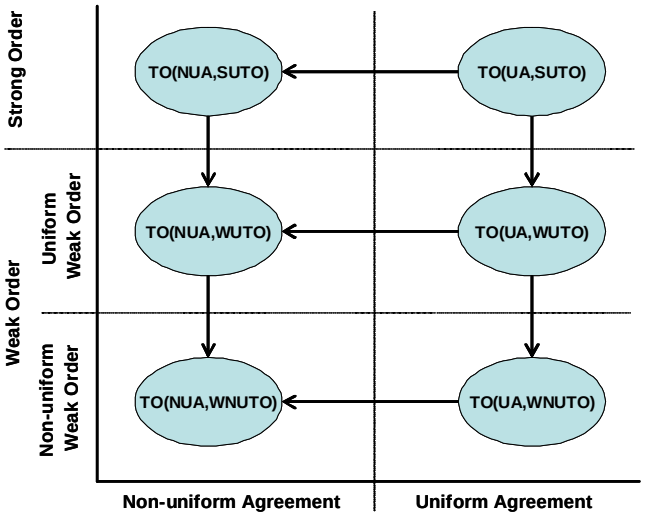

Fig. 1. A hierarchy of TO specifications

2.4 On the behavior of faulty processes

Each TO specification constrains all correct processes to deliver exactly the same ordered set of messages. Differences among the sequences of delivered messages delivered by faulty and correct processes can be characterized using the following patterns.

- EP1: a faulty process p delivers a prefix of the ordered set of messages delivered by correct processes;
- EP2: a faulty process p delivers some messages not delivered by correct processes;
- EP3: a faulty process p skips the delivery of some messages delivered by correct processes;
- EP4: a faulty process p delivers some messages in an order different from correct processes.

Each specification allows the occurrence of one or more of the above execution patterns. Moreover, from the definition of the $\rightarrow$ relation, it follows that for each pair of specifications $S, S' : S \rightarrow S'$, S' allows at least all execution patterns admitted by S. For example, $TO(UA, SUTO)$ allows EP1 while $TO(UA, WUTO)$ allows EP1 and EP3. Table 2 shows for each specification the admitted execution patterns. Let us note that these execution patterns are formally derived from specifications [4].

TO specification	Admitted execution patterns
$TO(UA, SUTO)$	EP1
$TO(UA, WUTO)$	EP1 or EP3
$TO(UA, WNUTO)$	EP1 or EP3 or EP4
$TO(NUA, SUTO)$	EP1 or EP2
$TO(NUA, WUTO)$	EP1 or EP2 or EP3
$TO(NUA, WNUTO)$	EP1 or EP2 or EP3 or EP4

Table 2. Possible differences between the behavior of faulty and correct processes

3 TO implementations in group communication systems

Group communication systems are one of the most successful class of systems implementing TO primitives. These systems adopt several distinct architectures [10]. For the sake of clarity, in the remainder of this paper we use a simplified architecture depicted in Figure 2, in which a Total Order layer implements a TO specification by relying on another layer, namely VSC, which provides virtually synchronous communications [11].[3] According to the virtual synchrony programming model, processes are organized into groups. Groups are *dynamic*, i.e. processes are allowed to join and voluntarily leave a group using appropriate primitives. Furthermore, faulty processes are excluded by groups after crashing. A *group membership service* provides each process of a group with a consistent *view* v_i composed by the identifiers of all non-crashed processes currently belonging to the group. Upon a membership change, processes agree on a new view through a *view change protocol*. At the end of this protocol, group members are provided with a view v_{i+1} that (i) is delivered to all the members of v_{i+1} through a view change event, and (ii) contains the identifier of all the members that deliver v_{i+1}. We consider a *primary component* membership service, e.g. [13], guaranteeing that all members of the same group observe the same sequence of views as long as they stay in the group. In this context, the VSC layer guarantees (i) that membership changes of a group occur in the same order in all the members that stay within the group, and (ii) that membership changes are totally ordered with respect to all messages sent by members. It is worth noting that the primary component membership service is not implementable in a non-blocking manner in asynchronous systems [14].[4]

The VSC layer also provides basic communication services. We consider two primitives, namely *Rcast* and *URcast*, which resembles non-uniform and uniform

[3] Let us remark that other approaches incorporating the implementation of *Order* and *Agreement* properties into a single protocol are possible, e.g. [12].

[4] In *partitionable* systems, groups may partition into subgroups (or *components*), e.g. due to network failures, and members of distinct subgroups can deliver distinct sequences of views. In this setting, specifying a total order primitive can drive to complex specifications, e.g. [2], whose usefulness has still to be verified [15]. However, non-blocking implementations of partitionable group membership services are feasible in asynchronous systems.

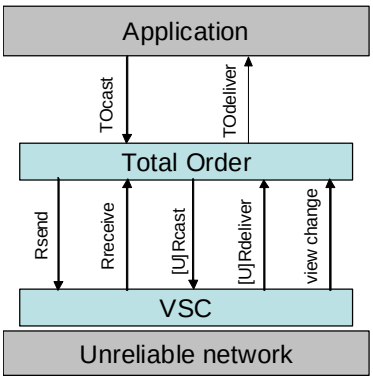

Fig. 2. Reference architecture of a group communication system

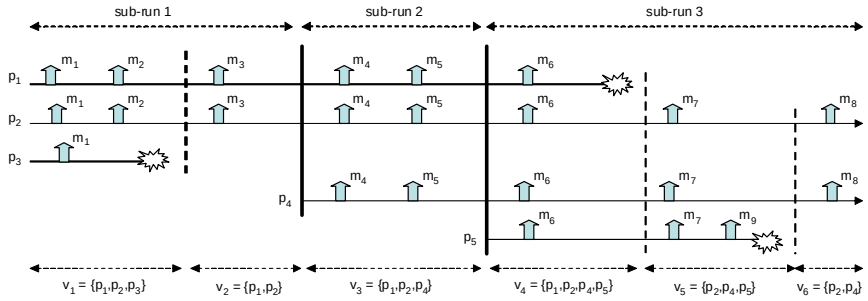

Fig. 3. A run of a system supporting dynamic process joins

reliable broadcast in the context of dynamic groups, respectively. These primitives ensure agreement of message deliveries for processes belonging to a view v_i and installing v_{i+1}, thus enforcing virtual synchrony. *URcast* also prevent faulty processes to deliver messages that will not be delivered by correct processes (i.e., it prevents the occurrence of EP2, in a way similar to *UA*). Interested readers can refer to [4] for a formal definition of these primitives.

3.1 Static vs. dynamic group communications

Following the model proposed by Hadzilacos and Toueg in [16], the properties introduced in Section 2.2 are based on a system model that does not take process joins into account. We now show how the TO specifications introduced in Section 2.3 can be used to classify also dynamic TO implementations, as the one given in the context of group communications. To this aim, we introduce the notion of *static sub-run*, i.e. a portion of the overall computation of a system supporting dynamic groups in which join events may *only* appear at the beginning of the

sub-run. Consider the computation depicted in Figure 3: it can be decomposed in three static sub-runs, namely sr_1, sr_2, sr_3. A sub-run can be described with events and process histories as those introduced in Section 2.1, i.e. $TOcast(m)$, $TOdeliver(m)$, and $crash$. As an example the sub-run sr_2 depicted in Figure 3 is composed by the histories of processes p_1, p_2 and p_4 containing the message delivery events of m_4 and m_5. Moreover, p_1 is correct in sr_1 and sr_2 while it is faulty in sr_3.

In this dynamic context, the TO specification enforced by a TO implementation $\mathcal{I}$ can be defined as follows.

Definition 1. Let $\mathcal{I}$ be a TO implementation and let $R_{\mathcal{I}}$ be the set of *static sub-runs* that $\mathcal{I}$ can generate. $\mathcal{I}$ enforces a TO specification S *iff*:

1. $R_{\mathcal{I}} \subseteq R_S$, and
2. $\forall S'\ S' \rightarrow S \Rightarrow R_{\mathcal{I}} \nsubseteq R_{S'}$.

Therefore the problem of finding the TO specification enforced by a TO implementation $\mathcal{I}$ boils down to find a TO specification S defining the smallest superset of $R_{\mathcal{I}}$.

As an example, in the run depicted in Figure 3, sub-runs sr_1 and sr_2 satisfy $TO(UA, SUTO)$, while sr_3 only satisfies $TO(NUA, SUTO)$ (due to p_5 delivering m_9). Therefore, an implementation $\mathcal{I}$ that may generate this run enforces at most $TO(NUA, SUTO)$ (i.e. $\mathcal{I}$ does not enforce $TO(UA, SUTO)$).

3.2 TO protocols

In this section we analyze the implementation of TO primitives offered by group communication systems. The most widely used protocols implementing the Total Order layer can be classified in *fixed sequencer* and *privilege-based* [7]. Interested readers can refer to [7] for a description of several other classes of TO implementations.

Fixed sequencer protocols In fixed sequencer protocols a particular process, i.e. the *sequencer*, is responsible for defining message ordering. This process is elected after each view change, usually on the basis of a deterministic rule applied to the current view, and defines a total order of messages by assigning to each message a unique and consecutive sequencer number. The sequence number assigned to a message is sent to all members, which deliver messages according to these numbers. These steps can be implemented using the following communication patterns.

- **Broadcast-Broadcast(BB).** The sender broadcasts message m to all members. Upon receiving m, the sequencer assigns a sequence number seq to m and then broadcasts seq to all members. As an example, the Ensemble system [8] implements this pattern;

- **Send-Broadcast(SB).** The sender sends message m to the sequencer, which assigns a sequence number seq and then broadcasts the pair $\langle m, seq \rangle$ to all members. This pattern is implemented, for example, by the Ensemble system [8];
- **Ask-Broadcast(AB).** The sender first gets a sequence number from the sequencer via a simple rendezvous, then it broadcasts the pair $\langle m, seq \rangle$ to all members. JavaGroups [9] is an example of a system implementing this pattern.

Privilege-based protocols In privilege-based protocols, a single logical token circulates among processes and grants to its holder the privilege to send messages. Each message is sent along with a sequence number derived from a value carried by the token which is increased after each message sent. Receiver processes deliver messages according to their sequence numbers. As only one token may circulate, and only the token holder may send messages, messages are delivered in a total order. Totem [17] and Spread [18] are examples of systems implementing this protocol.

In several privilege-based protocols, e.g. [17, 18, 9], processes are organized in a logical ring, and a process passes the token to the next process upon the occurrence of the first of the following internal events: (i) no more messages to send, or (ii) maximum use of some resources achieved (e.g. maximum token-holding interval, maximum number of messages sent by the process). These kind of protocols usually can be configured to implement $URcast$ at the Total Order layer, augmenting $Rcast$ with additional mechanisms thanks to the token passing. An example of such protocols is the one implemented by Spread [18].

Table 3 shows the TO specification enforced by each ordering protocol according to Definition 1 given in Section 3.1. In particular, for each protocol, we report the enforced TO specification depending on the used communication primitive, either $Rcast$ or $URcast$. Note that BB protocols are based on two broadcasts, which can be performed using different primitives of the VSC layer. The used communication primitives are reported in Table 3 in the form $first\_broadcast/second\_broadcast$. These results have been formally derived in [4], which also includes the pseudo-code of each algorithm.

4 Performance analysis

Typically, the cost in terms of performance of implementing a property increases with the strenght of the same property. For instance, implementing UA costs more than implementing NUA, as this requires to delay the delivery of messages within processes in order to be sure that they will be delivered by all correct processes. As a consequence, implementations of $TO(NUA, WNUTO)$ are likely to perform better than implementations of other specifications. In the remainder of this section we present a simple performance analysis of some TO

Ordering protocol	Communication primitive	TO specification
Broadcast-broadcast sequencer	$Rcast/Rcast$	$TO(NUA, WNUTO)$
	$URcast/URcast$	$TO(UA, SUTO)$
	$Rcast/URcast$	$TO(NUA, WUTO)$
	$URcast/Rcast$	$TO(UA, WNUTO)$
Send-broadcast sequencer	$Rcast$	$TO(NUA, WNUTO)$
	$URcast$	$TO(UA, SUTO)$
Ask-broadcast sequencer	$Rcast$	$TO(NUA, WUTO)$
	$URcast$	$TO(UA, SUTO)$
Privilege-based	$Rcast$	$TO(NUA, WUTO)$
	$URcast$	$TO(UA, SUTO)$

Table 3. TO specification enforced by each ordering protocol

implementations of real systems. To this end, we first introduce the group toolkits chosen for evaluation, namely Ensemble [8], Spread [18] and JavaGroups [9]. Then we report the experimental analysis we carried out on such systems.

4.1 Group communication toolkits

In this section we exploit the framework defined in the previous sections in order to identify the specifications enforced by TO primitives implemented in the considered group communication systems.

Spread. Spread is a toolkit designed for large scale networks based on a client-daemon architecture. It offers several communication abstraction, enabled by selecting the so-called "service type". Spread implements a partitionable membership service based on the *extended virtual synchrony model* [19], which extends virtual synchrony to partitionable environments. To comply with the reference architecture of Figure 2, it is thus necessary to assume either absence of network partitioning or the presence of a software filter implementing a primary component membership service and virtual synchrony on top of extended virtual synchrony [19]. In these cases, the privilege-based protocol embedded by Spread (enabled by selecting the `Agreed` service type) implements $TO(NUA, WUTO)$. In contrast, selecting the `Safe` service type, the protocol implements $URcast$ on top of *Rcast* (see Section 3.2) and thus the implemented TO specification is $TO(UA, SUTO)$.

Ensemble. Ensemble provides fine-grained control over its functionality, which can be selected simply layering *micro-protocols*, i.e. well-defined stackable components implementing simple and specific functions. In particular, Ensemble can be configured to implement virtual synchrony and a primary component membership service. A TO primitive is obtained layering a micro-protocol resembling the Total Order layer into a virtually synchronous stack. In the following we consider the micro-protocols named `Seqbb` and `Sequencer`, which correspond to BB and SB fixed sequencer protocols using *Rcast*, respectively (see Section 3.2). As shown in [4], layering one of these protocols in a virtually synchronous stack allows us to enforce $TO(NUA, WNUTO)$.

JavaGroups. JavaGroups is a Java group communication system based on the concept of micro-protocols (as Ensemble). As for Spread, JavaGroups does not exactly comply with our reference architecture, as it does not provide a primary component membership service. However, this can be implemented by coding a simple specific micro-protocol [19]. JavaGroups offers two micro-protocols implementing the Total Order layer, namely `TOTAL`, which embeds an AB fixed sequencer protocol using *Rcast*, and `TOTAL_TOKEN`, which embeds a privilege-based protocol enabled to implement *URcast* on top of *Rcast*. As proven in [4], these protocols enforce $TO(NUA, WUTO)$ and $TO(UA, SUTO)$, respectively, if JavaGroups is provided with the primary component membership service micro-protocol.

Tables 3 and 4 summarize the previous discussion. Let us remark that information given by Table 4 hold as long as systems are configured to implement the virtual synchrony model (and not the extended virtual synchrony model), which in some cases requires to extend the toolkit with additional software components, as discussed above (see Appendix A for further details on systems' configurations).

Let us finally note that none of the analyzed toolkit implements all six TO specifications (Ensemble supports other ordering protocols, but they are not able to enforce all remaining specifications).

Toolkit	TO implementation	Protocol type	TO specification
Spread	`Safe`	$PB(URcast)$	$TO(UA, SUTO)$
	`Agreed`	$PB(Rcast)$	$TO(NUA, WUTO)$
Ensemble	`Seqbb`	$BB(Rcast/Rcast)$	$TO(NUA, WNUTO)$
	`Sequencer`	$SB(Rcast)$	$TO(NUA, WNUTO)$
JavaGroups	`TOTAL_TOKEN`	$PB(URcast)$	$TO(UA, SUTO)$
	`TOTAL`	$AB(Rcast)$	$TO(NUA, WUTO)$

Table 4. Main characteristics of the group toolkits with respect to their TO implementations

4.2 Experimental settings.

Testbed environment. The testbed environment consists of four Intel Pentium 2.5GHz workstations that run Windows 2000 Professional. On each workstation Spread 3.17.0, JavaGroups 2.0.6 and Ensemble 1.40 have been installed and configured. The workstations are interconnected by a 100Mbit Switched Ethernet LAN.

Testbed application. All the experiments involve a static group of four processes, each running on a distinct workstation.[5]

We run a distinct experiment for each row of Table 4, in order to evaluate the performance of each TO implementation. Every experiment consists of ten failure-free rounds. During each round, every process sends a burst of B messages using the TO protocol under examination. Each message has a payload composed by the sender identifier and a local timestamp. The size of the payload is thus very small, i.e. about 8 bytes. After sending the burst, each process waits to deliver all of its messages and those sent by other members (i.e. it waits to deliver $T = 4 \times B$ messages). Each time a process delivers one of the messages sent by itself, it evaluates the message latency exploiting the timestamp contained in the payload. At the end of the experiment, each process evaluates the average message latency. Per-process average message latencies are further averaged to obtain a system message latency. Furthermore, we evaluate the overall system throughput, which is obtained as the sum of the throughput experienced by each process in the experiment. This is in turn calculated as the average number of messages delivered per second during each round. Results were obtained by letting the burst size B vary in $\{1, 10, \ldots 100\}$, repeating each experiment 10 times and averaging the results.

We decided to test group toolkits under bursty traffic as developers usually encounter problems in these settings [5].

4.3 Experimental results

Figure 4 and 5 show the overall comparison. In particular, Figure 4 depicts the average message latency, and Figure 5 presents the overall system throughput as a function of B.

The results can be evaluated under several aspects. In particular, the different behavior of the tested configurations depends on (i) the TO specification implemented by the configuration, (ii) the TO protocol used to implement that specification, and (iii) the way the protocol is implemented, which accounts for different architectures, optimizations, implementation language, etc.

Let us first analyze implementations enforcing the same specifications. Concerning JavaGroups (`TOTAL_TOKEN`) and Spread (`Safe`), which enforce $TO(UA, SUTO)$, this two configurations exploit a similar privilege-based protocol. Spread (`Safe`) outperforms JavaGroups (`TOTAL_TOKEN`), as the average message latency experienced with the latter configuration is about 2 to 3 times the one obtained with the former, while the overall system throughput suffers from a reduction of about 30% to 60%. A similar argument applies to JavaGroups (`TOTAL`) and Spread (`Agreed`), both implementing $TO(NUA, WUTO)$. In this case, the performance gain obtained with Spread's configuration is even more evident. For example, the latency experienced with JavaGroups is about 4 to 6 times the

[5] In the case of Spread, which adopts a daemon-based architecture, we run both a daemon and a client on each workstation.

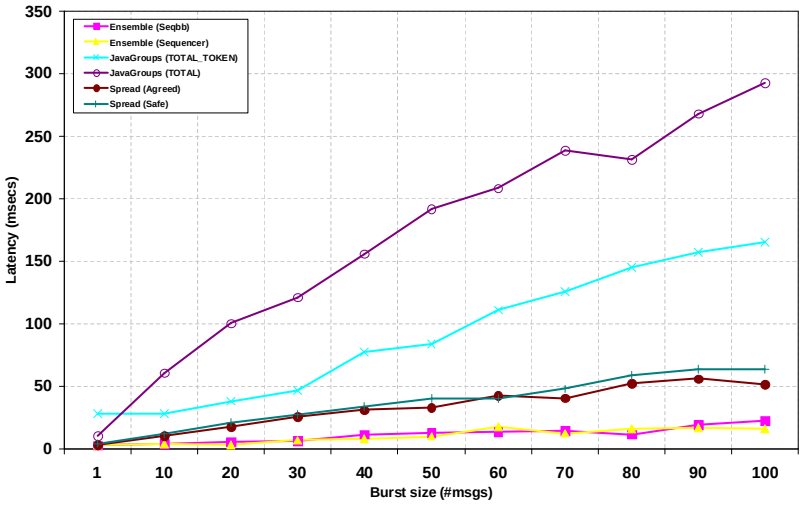

Fig. 4. Average message latency

one obtained with Spread. Finally, the two Ensemble configurations, both implementing $TO(NUA, WNUTO)$, perform almost the same.

From the above discussion, it is evident that the performance of a TO primitive enforcing a given TO specification substantially depends on the overall characteristics of the system implementing it. In fact, in spite of implementing the same protocol (and thus the same TO specification), Spread (`Safe`) and JavaGroups (`TOTAL_TOKEN`) gives substantially different performance. This is due to several factors, e.g. implementation language (C++ vs. Java), architectural design and optimizations.

A second step is therefore to compare different configurations of the same system, in order to avoid biases stemming from implementation issues. Concerning Spread's configurations, `Agreed` outperforms `Safe`. Differences are due to the increased amount of synchronization required by Spread (`Safe`) to enforce $TO(UA, SUTO)$. Very interestingly, the performance penalty paid by Spread (`Safe`) is small, both in terms of additional message latency (1.2 times the one of Spread (`Agreed`)) and in terms of throughput reduction (15% on average). In contrast, JavaGroups (`TOTAL_TOKEN`) outperforms JavaGroups (`TOTAL`), with the latter experiencing twice the average message latency and an average throughput reduction of 30% with respect to the former. These results are unexpected, as JavaGroups (`TOTAL`) implements a TO specification weaker than the one implemented by JavaGroups (`TOTAL_TOKEN`) (i.e. $TO(NUA, WUTO)$ vs. $TO(UA, SUTO)$). We argue that these results are due to JavaGroups (`TOTAL`) embedding an AB fixed sequencer protocol, which is not well suited for configurations in which processes frequently generate bursts of messages of small size. Furthermore, this algorithm suffers from the load to which the sequencer is

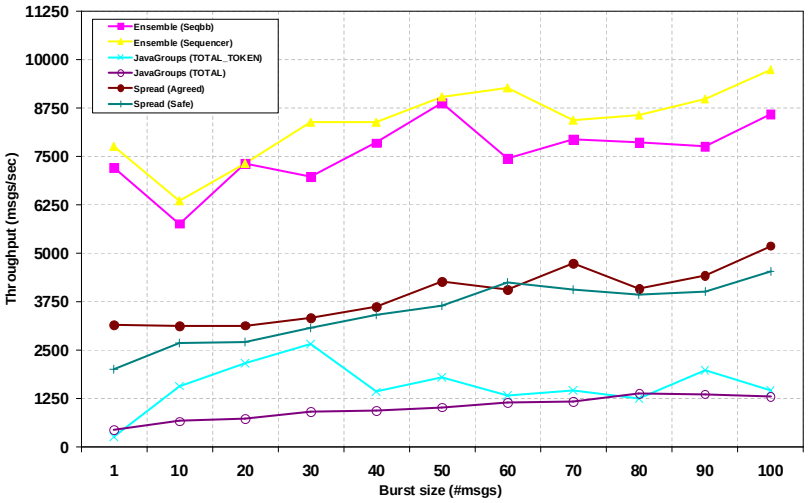

Fig. 5. Overall system throughput

subject during the experiments. In other words, the experimental settings seem to favor the privilege-based algorithm of JavaGroups (`TOTAL_TOKEN`), which is thus able to perform better, even though implementing a stronger specification.

A final note is on the two Ensemble configurations. In a setting providing hardware broadcast, as the one used for the experiments, BB and SB algorithms perform very similarly. Furthermore, these two configurations exhibit the best results, having the lowest message latency and the highest throughput. In particular, the average message latency experienced with Spread is about 2 to 5 times the one obtained with the two Ensemble configurations. This ratio roughly increases up to 15, if we consider JavaGroups. Concerning throughput, Spread's configurations exhibit a reduction of about 50% with respect to Ensemble's configurations. Considering JavaGroups, the throughput reduction is about 80%. These results can be explained noting that Ensemble's configurations implement the weakest TO specifications, i.e. $TO(NUA, WNUTO)$.

4.4 Discussion

The main contributions regarding performance of TO implementations either compare protocols using simulations and/or analytical studies, e.g. [20, 6] or deal with experiments done comparing several algorithms embedded into a single framework, e.g. [5]. We deem that this information does not fully enable immediate comparison of real systems, which is important from a developer's point of view. Upon building an application and thus having to match correctness and performance requirements, a comparison of TO protocols in a simulated environment or in a single real framework is not sufficient, especially in case the

developer wishes to select the best TO implementation choosing from a set of available ones. This set of experiments is aimed to complement the available information in order to facilitate this selection. Our plan is to provide a practitioner with a largest set of experiments in the near future, to cope with a larger set of application scenarios including actual failures.

5 Concluding remarks

This paper provided practitioners with a comprehensive yet quick and easy to understand reference for dealing with total order communications. Existing TO specifications have been classified into an hierarchy which actually models their differences in terms of admitted scenarios. Furthermore, six TO implementations provided by three freely-available systems have been analyzed, matching them against the hierarchy and comparing them from a performance point of view. On the basis of this information complemented with the one provided by simulations [20, 6, 5], practitioners will be able (i) to understand which TO specification meets their application's safety requirements, and (ii) to select the available TO implementation enforcing that TO specification while yielding the best performance. Concerning the latter issue, our experiments point out that the performance of a TO primitive is clearly dependent on the enforced specification, other than the employed algorithm and implementation specific details.

References

1. Schneider, F.B.: Replication Management Using the State Machine Approach. In Mullender, S., ed.: Distributed Systems. ACM Press - Addison Wesley (1993)
2. Chockler, G., Keidar, I., Vitenberg, R.: Group Communications Specifications: a Comprehensive Study. ACM Computing Surveys **33** (2001) 427–469
3. Powell, D.: Group Communication. Communications of the ACM **39** (1996) 50–97
4. Baldoni, R., Cimmino, S., Marchetti, C.: Total order communications over asynchronous distributed systems: Specifications and implementations. Technical Report 06/04, Università di Roma "La sapienza" (2004) Available at http://www.dis.uniroma1.it/∼midlab/publications.html.
5. Friedman, R., van Renesse, R.: Packing Messages as a Tool for Boosting the Performance of Total Ordering Protocols. Technical Report TR95-1527, Department of Computer Science, Cornell University (1995) Submitted for publication.
6. Dèfago, X.: Agreement-Related Problems: From Semi Passive Replication to Totally Ordered Broadcast. PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland (2000) PhD thesis no. 2229.
7. Dèfago, X., Schiper, A., Urbán, P.: Total order broadcast and multicast algorithms: Taxonomy and survey. Technical Report IC/2003/56, École Polytechnique Fédérale de Lausanne, Switzerland (2003)
8. Hayden, M.: The Ensemble System. PhD thesis, Cornell University, Ithaca, NY, USA (1998)
9. Ban, B.: Design and implementation of a reliable group communication toolkit for java. Cornell University (1998)

10. Mena, S., Schiper, A., Wojciechowski, P.: A step towards a new generation of group communication systems. Technical Report IC/2003/01, École Polytechnique Fédérale de Lausanne, Switzerland (2003)
11. Birman, K., Joseph, T.: Exploiting Virtual Synchrony in Distributed Systems. In: Proceedings of the 11th ACM Symp. on Operating Systems Principles. (1987) 123–138
12. Chandra, T.D., Hadzilacos, V., Toueg, S.: The weakest failure detector for solving Consensus. Journal of ACM **43** (1996) 685–722
13. Birman, K., van Renesse, R.: Reliable Distributed Computing with the ISIS toolkit. IEEE CS Press (1994)
14. Chandra, T., Hadzilacos, V., Toueg, S., Charron-Bost, B.: On the Impossibility of Group Membership. In: Proc. of the 15th ACM Symposium of Principles of Distributed Computing. (1996)
15. Schiper, A.: Dynamic group communication. Technical Report IC/2003/27, École Polytechnique Fédérale de Lausanne, Switzerland (2003)
16. Hadzilacos, V., Toueg, S.: Fault-Tolerant Broadcast and Related Problems. In Mullender, S., ed.: Distributed Systems. Addison Wesley (1993)
17. Amir, Y., Moser, L., Melliar-Smith, P.M., Agarwal, D., Ciarfella, P.: The Totem single-ring ordering and membership protocol. ACM Transactions on Computer Systems **13** (1995) 93–132
18. Amir, Y., Stanton, J.: The Spread Wide Area Group Communication System. Technical Report CNDS-98-4, Center for Networking and Distributed Systems, Computer Science Department, Johns Hopkins University, 3400 N. Charles Street Baltimore, MD 21218-2686 (1998)
19. Moser, L.E., Amir, Y., Melliar-Smith, P.M., Agarwal, D.A.: Extended Virtual Synchrony. In: Proceedings of the 14 International Conference on distributed Computing Systems. (1994)
20. Cristian, F., de Beijer, R., Mishra, S.: A Performance Comparison of Asynchronous Atomic Broadcast Protocols. Distributed Systems Engineering Journal **1** (1994) 177–201

Appendix A: Group toolkit configuration

This appendix gives additional details on how to configure the systems analyzed in the paper to use their TO primitives (see Table 5). For further details, interested readers are referred to systems' reference manuals.

Spread. Developers must simply label messages to enact Spread's services. In particular, the `Agreed` label enables the Spread (`Agreed`) configuration, whereas the `Safe` label triggers the Spread `Safe` configuration. There is no need for further configurations. However, developers must provide an implementation of virtual synchrony and primary component membership service filters to achieve the TO specifications described in Table 5.

Ensemble. In Ensemble each process has to specify the stack to use upon joining the group. This can be done either by specifying desired *properties* (which identify portions of protocol stacks), or by directly selecting the micro-protocols. In

Toolkit	Configuration	Additional mechanisms
Spread (Safe)	`Safe` service type	VS + PC GMS filters
Spread (Agreed)	`Agreed` service type	VS + PC GMS filters
Ensemble (BB)	VS + PC GMS + `Seqbb`	-
Ensemble (SB)	VS + PC GMS + `Sequencer`	-
JavaGroups (TB)	VS + `TOTAL_TOKEN`	PC GMS filter
JavaGroups (AB)	VS + `TOTAL`	PC GMS filter

Table 5. Configurations and additional mechanisms necessary to achieve TO specifications supported by each of the examined group toolkits

both cases, it is necessary to set a particular field in the data structure representing the so-called join options. In the first case, the `properties` string should be set. The string

```
Gmp:Sync:Heal:Frag:Suspect:Flow:Slander:Total:Primary
```

allows to achieve a TO primitive enforcing $TO(NUA, WNUTO)$ by means of an ordering protocol in a stack also providing virtual synchrony and a primary component membership service. This configuration automatically selects `Seqbb` as the ordering protocol. To use a different protocol, it is necessary to explicitly select all the protocols of the stack. In this case, the `protocol` string should be set, e.g. to

```
Top:Heal:Primary:Present:Leave:Inter:Intra:Elect:Merge:Slander:Sync:Suspect:Stable:Vsync:
Frag_Abv:Partial_appl:Seqbb:Collect:Frag:Pt2ptw:Mflow:Pt2pt:Mnak:Bottom
```

which corresponds to the previous string of properties. To use other TO protocols, it is necessary to substitute `Seqbb` with the protocol to be used, e.g. `Sequencer`, in the string above.

JavaGroups. Also in JavaGroups the protocols composing the stack can be specified through a string. As an example, the string

```
UDP:PING:FD_SOCK:VERIFY_SUSPECT:STABLE:NACKACK:UNICAST:FRAG:TOTAL_TOKEN:FLUSH:GMS:QUEUE
```

represents a stack providing a total order through the `TOTAL_TOKEN` protocol. Instead, the string

```
UDP:PING:FD_SOCK:VERIFY_SUSPECT:STABLE:NACKACK:UNICAST:FRAG:FLUSH:GMS:TOTAL:QUEUE
```

can be used to exploit the `TOTAL` protocol. However, developers have to implement a primary component membership service filter, which has to be inserted into the stack in order to achieve TO primitives compliant with $TO(UA, SUTO)$ and $TO(NUA, WUTO)$, respectively.