Flexible and Reliable Messaging using Runtime Monitoring

Sylvain Hallé
University of California
Santa Barbara, CA, USA
Email: shalle@acm.org

Roger Villemaire
Université du Québec à Montréal
Montréal, Canada
Email: villemaire.roger@uqam.ca

***Abstract*—The asynchronous nature of communications in message-based systems like service-oriented architectures introduces two major issues: inability to detect lost and out-of-sequence messages, and unrealizability of some messaging protocols. We show that these problems are actually different manifestations of the same phenomenon: communicating peers ending up with divergent views of the message exchange in which they are involved. We introduce the concept of monitor-based messenger (MBM), which processes messages locally through a runtime monitor enforcing a specific protocol of interaction, and stamps them with a monitoring token. We demonstrate that: 1) some unrealizable protocols become realizable using MBMs; 2) MBMs offer protection against unreliable messaging, and can decrease delivery time and required queue size compared to strict messaging solutions.**

***Keywords*-messaging; runtime monitoring; middleware; asynchronous communications;**

I. Introduction

An increasing number of systems rely on message-based exchanges for their means of communication: for example, the Service-oriented Architecture (SOA) relies mostly on the exchange of SOAP messages through standard communication protocols. In most cases, the communication scheme employed is asynchronous, and allows a sent message to be picked by the receiver at a later time from a local receiving queue.

While the asynchronous mode simplifies communication in situations where blocking is not necessary, it introduces two new problems. The first one is *unreliable messaging*: there is no way for the receiver to detect that a message never made it to its destination, or that messages in the receiving queue were actually sent in a different order. Second, two communicating peers following the same protocol can deadlock or produce non-compliant traces of messages, even in the case of perfect channels and infinite queue sizes: this is called *unrealizability*.

In Section II, we describe these seemingly separate issues and observe that they are actually different manifestations of the same phenomenon: communicating peers ending up with divergent "views" of the message exchange in which they are involved. Consequently, we argue that keeping track of the state of the conversation on each peer, using some form of runtime monitoring, and sharing this information to uncover any discrepancies, should help alleviate both of these problems.

There already exists a large body of work on runtime monitoring which can be put to good use. A *runtime monitor* eavesdrops the messages sent and received by a given peer, and makes sure that the sequence of such messages follows a set of constraints, called a *protocol*. In Section III, we introduce *monitor-based messengers* (MBMs), which locally stamp each outbound message with a token depending on their state. We first show how divergent views of the same protocol can be detected by communicating peers; in some cases, their views can even be made to re-converge to a common state of the protocol, without the need for a centralized mechanism. This entails that there exist unrealizable protocols which become realizable with MBMs.

An interesting side effect of this property is that desynchronizations can also be *tolerated* on purpose when reconvergence of the protocol views is inevitable. In Section IV, we show how this behaviour makes MBMs robust against out-of-sequence messages, and can actually decrease their waiting time in receiving queues. To illustrate our point, we describe in Section V the MBMonitor, a messaging layer in Java that implements these principles. Our results indicate that, for arbitrarily shuffled message sequences, reliable messaging based on runtime monitoring can decrease delivery time and required queue size.

II. Divergent Views in Message-Based Systems

We take a *protocol* as a specification of a global pattern of sequences of messages. These sequences can be thought as being taken by a global observer, writing down each message as it is *sent*. Protocols are common place in service oriented architectures, where they are called *choreographies* [1] or *conversations* [2]. They can, for example, represent constraints on the way a service can be accessed.

A. Asynchronous Communications and Message Protocols

To formalize the notion of protocol, a formal model of message-based communications is required. We concentrate our study on bidirectional communications between exactly

We gratefully acknowledge the financial support of the Natural Sciences and Engineering Research Council of Canada (NSERC) and the Fonds québécois de recherche sur la nature et les technologies (FQRNT).

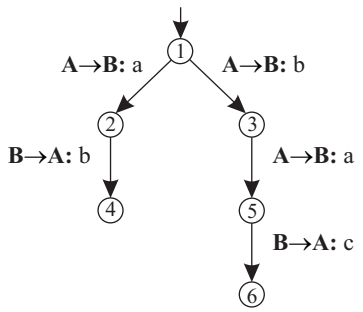

Figure 1. A simple protocol specification

two peers. We view each peer as a system composed of a finite-state control and a message queue. A transition between two control states q and q' can be of two forms: $q \rightarrow_{!m} q'$ indicates that some message m must be sent (to the only other peer), while $q \rightarrow_{?m} q'$ indicates that the control state can go from q to q' if m is the first message in the receiving queue. In such a case, the message is removed from the queue.

A *channel system* $\mathcal{L}$ is a state transition system composed of a set of such peers. The global state of a channel system is the unique combination of each peer's control state and queue contents. Transitions from one global state to another are the result of exactly one peer either sending or receiving a message. This channel system represents asynchronous communications, since the send and receive events in each channel can be arbitrarily far apart in time.

A *protocol* is a description of admissible sequences of *send* events in a channel system. It can be represented by a special case of finite-state machine called a guarded automaton. Each state of the automaton represents a state in the protocol; a transition between two states represents the sending of a message, and is augmented by logical expressions, called *guards*, which must evaluate to true for the transition to be taken. For the sake of simplicity, we omit guards in our analysis; their introduction is straightforward. Figure 1 shows an example of a protocol specification between two peers, A and B. Each transition is noted $\mathbf{X} \rightarrow \mathbf{Y} : m$, denoting that X sends message m to Y.

Definition 1. *A protocol specification $\mathcal{C}$ is a tuple $\langle S, s_0, F, M, \delta \rangle$ where S is a set of control states, $s_0 \in S$ is the initial state, $F \subseteq S$ is a set of final or accepting states, M is a set of messages, and $\delta : S \times M \rightarrow S$ is a transition* function.

The transition δ is assumed to be a partial function; it maps any pair of state and message to at most one control state; we write $\delta(s, m) = \emptyset$ to indicate that message m cannot be sent or received from state s. We can now define what it means for a channel system to follow a protocol specification:

Definition 2. *A channel system $\mathcal{L}$ follows a protocol specification $\mathcal{C}$ if every global trace of $\mathcal{L}$, trimmed of its receive events, is a trace of $\mathcal{C}$.*

Examples of message-based protocols abound in the literature for various contexts. The IBM Conversation Support [3] provides a library of protocol templates for various business activities. RosettaNet [4] defines 107 predefined patterns of interaction between business partners, called Partner Interface Processes (PIP); Figure 2 shows a portion of an exchange between a buyer and a seller, following such a PIP. Moreover, any specification expressed using, for example, UML Message Sequence Charts [1] or web service choreography languages such as WS-CDL [5] are by definition finite, message-based protocols which can be translated into the above form, and analyzed through the techniques presented in this paper.

However, the interplay of asynchronous communications and protocol specifications is the source of two issues, which we now describe.

B. Unrealizable Protocols

Because of the nature of asynchronous communications, it is possible that two peers end up in deadlock, or generate a global message trace outside the specification. This can be the case, even when both peers individually follow the same protocol specification.

Figure 1 shows an example of such a protocol. Both peers A and B have the choice of starting the exchange of messages. However, if each peer makes this choice without knowledge of the other's decision, the following situation can occur:

1) B sends message *b* to A
2) Before *b* reaches A, A sends *a* to B
3) The messages cross over the communication link, and eventually reach their destination, with A mistakenly believing it is initiating the conversation
4) B wrongly assumes the *a* it receives is in response to *b*, and replies with *c*

The global trace of messages sent, *bac*, is not part of the protocol and constitutes a violation. However, the problem does not lie in wrongdoing from any of the peers: neither one took an illegal transition, given their local state and the content of their queues. The problem rather stems from the protocol itself, which provokes such unexpected behaviours when asynchronous communications are used. A protocol spared of these side effects exhibits a property called *realizability*:

Definition 3. *A protocol specification $\mathcal{C}$ is realizable if the possible interactions of each peer produces exactly all the send traces of $\mathcal{C}$.*

One can see that the protocol in Figure 1 is not realizable. More precisely:

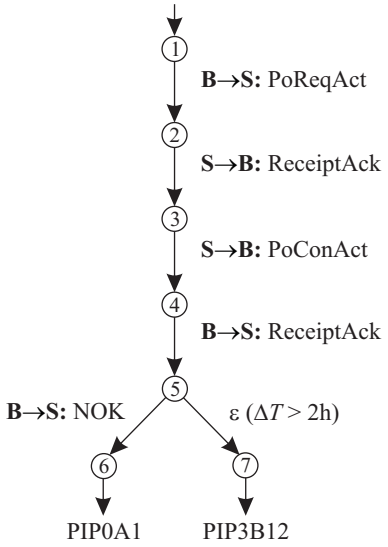

Figure 2. A portion of a Partner Interface Process from RosettaNet

Observation 1. *The protocol in Figure 1 is* unrealizable*: when peer A reaches state and peer B reaches state 3, an invalid global sequence of messages will be produced.*

A concrete example of an unrealizable protocol is provided by [6]; by studying the RosettaNet PIP shown in Figure 2, it shows that a failure to deliver a valid message within its time constraint can cause mutually conflicting views of an interaction. The problem arises when the seller sends a *NOK* message in time, but that *NOK* is either lost or reaches the seller after the two-hour limit. In such a case, buyer and seller will continue their exchange of messages, in two different contexts.

Current solutions basically amount to statically analyze the protocol to determine if it is realizable. In [7], an algorithm is presented to generate from a protocol local patterns of interactions for each peer; if the protocol is realizable (or *locally enforceable*), the composition of these patterns will produce exactly the desired behaviours.

However, there currently does not exist necessary and sufficient conditions for a protocol to be realizable; it is not even known whether the general problem is decidable. [8] provides a set of sufficient conditions for realizability; these conditions can be statically analyzed on the protocol; yet, the authors remark that there exist real-world protocols which fail these realizability conditions.

C. Unreliable Messaging

Since they impose constraints on the sequencing of messages, protocols are also sensitive to imperfect communications, where messages can be received out of sequence or lost. This can be due to different messages taking different paths in the network, variable network latency or processing time in some of the nodes along the path, a severed communication link or even server downtime.

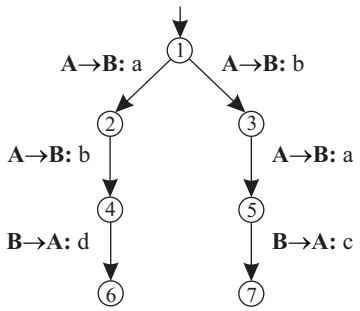

Figure 3. A protocol sensitive to message shuffling

Figure 2 can be seen as a protocol sensitive to unreliable messaging, since various message transmission times can take the receiver to different next states. As an additional example, Figure 3 shows how the sequence of messages *ab*, sent through an unreliable link, can be shuffled and arrive at its destination as the sequence *ba*, impacting B's reply.

Hence, unreliable messaging will have the same consequence as for realizability: messages received out of sequence can take two peers on different states of the same protocol.

Observation 2. *The protocol in Figure 3 is sensitive to* unreliable messaging*: if messages are received out of sequence, there exists a message trace which can take peer A to state 4 and peer B to state 5.*

There exist approaches which, although not directly related to the present problem, bear some similarities. *Transaction processing systems*, most prominent in databases, focus on the atomicity of a sequence of operations, but not necessarily to the order in which they are performed; two- and n-phase commit protocols require a centralized coordinator and additional messages in order to do so. *Virtual synchrony* [9] is used to synchronize state information among distributed members of a group; however, it focuses on broadcast messages and not one-to-one communications.

In [10], various architectures are enumerated to cope with unreliable messaging. For example, the application itself can take care of reliable messaging, awaiting confirmation of reception and retransmitting messages through various means. Another solution is to add a messaging layer between the application and the communication link; this so-called message-oriented *middleware* interfaces with the application and transparently ensures that messages are correctly relayed to their intended receiver.

For example, to communicate through the WS-ReliableMessaging (WS-RM) protocol [11], the sender first opens a *sequence* on the receiving end; this sequence is acknowledged by the receiver. The sender can then transmit messages to the receiver, adding to each SOAP message an additional WS-ReliableMessaging header. This header

contains a sequence number that increments by 1 for every new message sent. This operation ensures that any lost messages are detected and can be retransmitted through appropriate acknowledgements. In addition, a sequence can be opened with the property that the message ordering has to be respected; guided by the unique sequence numbers stamped with each message, the receiver can temporarily queue out of sequence inputs, and relay to the application (albeit with a potential delay) the same sequence that was sent. There exist a variety of other reliable messaging middleware and protocols, mostly using similar principles. Some of them only take care of detecting lost messages; this is the case of HTTPR [12], an extension of the standard HTTP protocol which allows additional payload information to ensure that each message is either delivered exactly once, or correctly reported as missing. Some of them can provide guarantees on both ordering and loss, such as IBM WebSphere MQ[1] or Microsoft Message Queuing.[2]

Reliable messaging has also spawned a fair amount of academic work. Systems based on *harmonized messaging* [13] and HCM3 [14] require a central coordinator for all peers to ensure proper delivery of messages. Other projects on reliable protocols guarantee message delivery by periodically retransmitting messages for which no acknowledgement has been received [15]–[17]; inversely, some others guarantee that all messages are received in the exact order they are sent [18].

III. Monitor-Based Messaging

The key point in the observations we previously made is that both messaging issues are actually a consequence of the same phenomenon: the possibility for two peers to reach divergent states in their protocol. This section centers on the idea of using a monitor to stamp outgoing messages with tokens based on the state of some protocol. It distinguishes itself from previous work in that: 1) no central coordinator is required; 2) all message sequences preserve the property of being *protocol compliant*, which in many cases is a looser constraint than exact ordering assumed by existing approaches.

Definition 4. *A* monitor *is a tuple $\mathcal{A} = \langle Q, q_0, \delta \rangle$ where: Q is a set of* states*; $q_0 \in Q$ is the initial state; $\delta : Q \times M \to Q$ is the transition or* update *function from a state and a message to another state.*

Formally, a monitor is a special case of finite-state automaton. The monitor starts in its initial state q_0; then, for each message m that is monitored, the update function $\delta(q, m)$ is called to take the monitor into its updated state q'. As such, a monitor only follows the conversation without doing anything about it; it is then up to its user to attach a meaning to its different states. A first application of such a model is to detect non-compliance of a messaging peer to a protocol specification. To this end, it suffices to define a function $f : Q \to O$ returning a value from a set of possible "outcomes" O for each state $q \in Q$ of the monitor. One of these outcomes, labelled $\perp$ (meaning "fail"), indicates that the protocol has been violated; another outcome, labelled $\top$ (meaning "OK"), indicates that the protocol has not (yet) been violated. This is what was done in [19].

A. Monitor-Based Messenger

In general, a runtime monitor is placed at the interface between each communicating peer and the outside world, similarly to messaging middleware. It is therefore ideally located to perform auxiliary functions in addition to its enforcement purpose: its knowledge of the interaction protocol can be put to use to help protocol-based reliable messaging between communicating peers.

Rather than simply associating an outcome to each monitor state, we define a tokenizing function $\tau : Q \to T$, which associates to a monitor state $q \in Q$ a symbol $t \in T$ called a *monitor token*. We then define the token outcome function $f : T \times T \to O$ as a mapping between *pairs* of monitor tokens and some outcome. A *stamping* function $\sigma : M \times T \to M$ is a function that takes a message and a monitor token and returns a new message with the stamp. The stamp can be read using $\hat{\sigma} : M \to T$, and can be removed using $\sigma' : M \to M$.

In the following, for M a set of messages, we denote by M^* the set of finite sequences of messages from the alphabet M; such a sequence $m_0, m_1, \ldots, m_n$ is written $\overline{m}$. Given a message $m \in M$, $\overline{m} \cdot m$ (resp. $m \cdot \overline{m}$) designates the concatenation of m at the end (resp. beginning) of $\overline{m}$; $\overline{m}^1$ designates the sequence of messages $m_1, m_2, \ldots, m_n$.

We write $m \in \overline{m}$ to indicate that there is a $i \geq 0$ such that $m = m_i$; the notation $\overline{m} - m$ designates the sequence of messages $\overline{m}'$ identical to $\overline{m}$, but where the first occurrence of m has been removed. Similarly, for $\overline{m}' = m\overline{m}''$, we define recursively $\overline{m} - \overline{m}' = (\overline{m} - m) - \overline{m}''$. Such notation allows $\overline{m}$ to be also manipulated as a multiset of elements in M.

Equipped with these tokenizing and stamping primitives, we can build a *monitor-based messenger* (MBM):

Definition 5. *An MBM is a tuple $\langle \mathcal{A}, S, \tau, f, \Delta \rangle$ where: $\mathcal{A} = \langle Q, q_0, \delta \rangle$ is a runtime monitor, $S \subseteq Q \times M^* \times M^* \times M^*$ is a set of states, τ is a tokenizing function, f is a token outcome function, and Δ is a transition function.*

A state of the messenger is a tuple $(s, \overline{q_{in}}, \overline{q_{out}}, \overline{w_{in}}') \in Q$, where s is a monitor state, $\overline{q_{in}}$ is an incoming message queue, $\overline{q_{out}}$ is an outgoing message queue. In addition, $\overline{w_{in}}$ is a multi-set of messages received by the messenger, but not yet relayed to the application layer. The transition relation is then defined as follows:

[1] http://www-3.ibm.com/software/mqseries/messaging

[2] http://www.microsoft.com/msmq

Definition 6. *The transition function* $\Delta : Q \times M \cup \{\epsilon\} \times \{in, out\} \to Q$ *is defined as follows:* $(s', \overline{q_{in}}', \overline{q_{out}}', \overline{w_{in}}') \in \Delta((s, \overline{q_{in}}, \overline{q_{out}}, \overline{w_{in}}'), m, a)$ *if and only if:*

1) $m = \epsilon$, $\overline{w_{in}}' = \overline{w_{in}}$, *and either*
   a) $\overline{q_{in}}' = \overline{q_{in}}^1$, $\overline{q_{out}}' = \overline{q_{out}}'$, $s' = s$, *or*
   b) $\overline{q_{out}}' = \overline{q_{out}}^1$, $\overline{q_{in}}' = \overline{q_{in}}'$, $s' = s$
2) $m \neq \epsilon$, $s' = s$, $\overline{q_{in}}' = \overline{q_{in}}$ *and either*
   a) $a = out$, $\overline{q_{out}}' = \overline{q_{out}} \cdot \sigma(m, \tau(s'))$, $\overline{w_{in}}' = \overline{w_{in}}$, $s' \in \delta(s, m)$ *and* $f(\hat{\sigma}(m), \hat{\tau}(\overline{w_{in}})) = \top$
   b) $a = in$, $\overline{w_{in}}' = \overline{w_{in}} \cup \{m\}$, $\overline{q_{out}}' = \overline{q_{out}}$ *or*
3) $m = \epsilon$, *there exists* $m' \in w_{out}$ *such that* $s' \in \delta(s, m')$, $\overline{q_{out}}' = \overline{q_{out}} \cdot \sigma'(m')$, $\overline{w_{in}}' = \overline{w_{in}} - \{m'\}$, $o(\hat{\sigma}(m'), s') = \top$

Informally, the transition relation works as follows. The first message in the incoming or outgoing queue can be removed at any time (cases 1a and 1b). If a message is to be sent, the messenger checks whether from its current monitor state, it represents a valid transition; if it is the case, the monitor state is updated, the message is stamped with a monitor token for that new state, and appended to the output queue —the other queues and sets remain unchanged (case 2a). If a message is to be received, it is simply put in the waiting multi-set of incoming messages (case 2b).

Finally, for any message m' in the waiting multi-set, if consuming m' results in a valid transition to a state s' in the runtime monitor, and that s' is compatible with the monitor token in m', then m' is removed from the waiting multi-set and appended to the incoming message queue with its monitor token removed (case 3).

This model is a generalization of the classical runtime monitor. Indeed, defining the stamping function as $\sigma(m) = m$ for all m, the tokenizing function as $\tau(s) = \epsilon$ the empty string for all states s, and the outcome function $f(s, s') = \top$ if and only if $s = s'$, the monitor-based messenger's behaviour reverts to a classical runtime monitor as defined in [19].

B. Consequences on Realizability

Obviously, a monitor-based messenger can hence perform runtime monitoring, by raising an error when a message received or to be sent is forbidden in the current state of the protocol. However, by redefining the monitor tokens and outcome functions, such a messenger can accomplish an additional function.

Definition 7. *Define* $T = Q$ *and* $\tau(q) = q$*, i.e. the set of monitor tokens is exactly the set of its states. Define the outcome function* $f_1(q, q') = \top$ *(meaning "OK") if and only if* q' *is reachable from* q *in* $\mathcal{C}$*, and* $\bot$ *(meaning "fail") otherwise.*

Following this definition, each peer stamps its outgoing messages with the current monitor state. By doing so, each peer can effectively "tell" its partner what state of the conversation it believes is the global state of the protocol. A problem arises whenever two peers fork and follow different paths in the protocol. Since communications are asynchronous, this can be detected as soon as one of the peers realizes that, from its current state, it could never catch up with the state of its partner. This is equivalent to the outcome function returning $\bot$ for a pair of states.

Stamping messages with additional information to prevent desynchronizations of some sort was suggested by [20], where a timestamp from the sender's local clock was used to keep its recipient's clock synchronized. The goal of this approach was to provide a total ordering of events in the absence of a global timekeeping mechanism. However, the approach assumes reliable messaging and bounds on message delivery time; moreover, it requires that additional synchronization messages be sent to all peers involved in a communication. Our approach rather takes protocol states, instead of time, as the *partial* ordering that is used as a stamp.

Going back to the unrealizable protocol in Figure 1, we show how unrealizable sequences can be detected. Suppose each peer mistakenly believes it initiates the conversation. Then A sends *a* to B, while B sends *b* to A, as previously. However, the message *a* is now stamped with the monitor token 2, while the message *b* is stamped with the monitor token 3. Since state 2 is not reachable from 3 (and vice versa), then $f_1(2, 3) = f_1(3, 2) = \bot$, and both peers will realize, upon receiving *b* (respectively *a*) that their views of the protocol are divergent.

If we assume perfect communication between the peers, we can show that this messenger can prevent global traces violating the protocol from stretching too long. Once two peers diverge on their view of a protocol, the erroneous sequence of messages they each send ends as soon as they receive a new message from their partner. This is the best that can be done without a central synchronization mechanism, since the information piggybacks the normal flow of messages.

We show that MBMs are strictly "safer" messengers than traditional FIFO queues, in that some unrealizable protocols become realizable with MBMs:

Theorem 1. *Let* $\mathcal{P}_{FIFO}$ *be the set of realizable protocols with classical, perfect FIFO messengers, and* $\mathcal{P}_{MBM}$ *be the set of protocols realizable with perfect MBMs. Then* $\mathcal{P}_{FIFO} \subset \mathcal{P}_{MBM}$.

Proof: If a protocol is realizable with FIFO messengers, it is trivially realizable by an MBM with an "empty" monitor with a single state s; the outcome function becomes $f(s, s) = \top$, and the messenger always dispatches messages as soon as they are received.

To show that the inclusion is strict, we must exhibit a protocol $\mathcal{C}$ unrealizable with FIFO messengers and realizable with MBMs. Figure 1 shows such a witness. We already

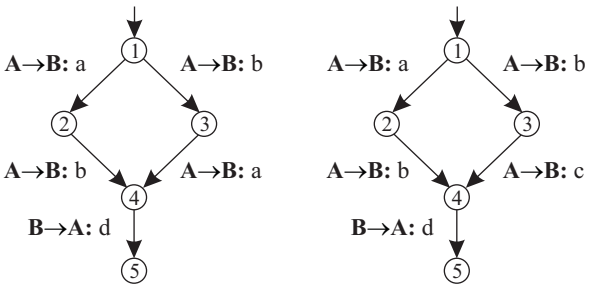

Figure 4. Two simple protocol specifications

know that $\mathcal{C}$ is unrealizable with FIFO messengers. For every trace in $\mathcal{C}$, there exists a global run where A and B behave synchronously: a message sent by A is received by B in the next step, and no messages cross over the wire. In such a case, each of their MBM will follow the same sequence of states in their runtime monitor, and the trace is hence realized.

Conversely, we must show that no global trace outside $\mathcal{C}$ can be produced. The only possible such trace is *abc*; however, for *c* to be transmitted by peer B, its MBM must first be in state 3 and receive *a* from peer A. In addition, the monitor token t on message *a* must be such that $f_1(3,t) = \top$. By definition of f_1, this is only possible if $t = 5$ or $t = 6$. We exclude $t = 6$, since it depends on B sending *c*, which has not happened yet. Therefore $t = 5$. Hence, when sending *a*, peer A was in state 3: it had already received B's message, and therefore the global trace is *bac*. ■

An important element is that for a protocol to be realizable, it must be possible to produce every global trace of messages. Our MBMs allow every trace of the witness protocol to be possibly realized. This is different from approaches that merely trim a protocol to a realizable subset of its behaviours (for example, the branch 1-2-4) to ensure no undesirable trace can ever be produced.

IV. Managing Desynchronizations

MBMs hence provide a safer context for enacting messaging protocols. However, up to now, the only possible way for two peers to continue a conversation is to follow the same sequence of visited states in their monitor. If a divergence is detected, it is assumed that the conversation cannot be extended by sending new messages. However, consider the protocol in the left part of Figure 4. Again, the situation where both A and B believe they initiate the exchange will be detected, with A reaching state 2, B reaching state 3, and $f_1(2,3) = f_1(3,2) = \bot$. Yet, in this case, this conflicting view has no harmful consequence, since both *ab* and *ba* are valid sequences which eventually converge back to state 4.

A. MBMs with History

It is therefore desirable to relax the definition of the outcome function f_1, to allow divergent paths in the monitor to be taken, as long as they eventually "amount to the same thing". To this end, from a protocol specification $\mathcal{C} = \langle S, s_0, F, M, \delta_{\mathcal{C}} \rangle$, let us define a monitor $\mathcal{A}_{\mathcal{C}} = \langle Q, q_0, \delta_{\mathcal{A}_{\mathcal{C}}} \rangle$ where the set of states is $Q \subseteq S^*$. Therefore, the states of $\mathcal{A}$ are (finite) sequences of states of $\mathcal{C}$; let $q_0 = s_0$. For a message m, a state of $\mathcal{C}$ $s' \in S$ and two states of $\mathcal{A}$ $q = s_1, \dots s_k$ and q', we have that $q' \in \delta_{\mathcal{A}_{\mathcal{C}}}(q, m)$ if and only if $q' = q \cdot m$ and $s' = \delta_{\mathcal{C}}(s_k, m)$.

The monitor $\mathcal{A}_{\mathcal{C}}$ is therefore a runtime monitor of the protocol $\mathcal{C}$, but such that each state keeps track of the history of previously visited states. For two traces $q = s_1, \dots s_k$ and $q' = s'_1, \dots s'_\ell$, we define the last *synchronization point* as the highest $n \geq 0$ such that $s_n = s'_n$. Define $r = s_n, s_{n+1}, \dots, s_k$ and $r' = s'_n, s'_{n+1}, \dots, s'_\ell$ the desynchronized suffixes of each token.

Given a sequence of states $s_1, \dots, s_n$, a possible message trace is a sequence of messages $m_1, \dots, m_{n-1}$ such that for every $1 \leq i \leq n-1$, $\delta(s_i, m_i) = s_{i+1}$. Intuitively, it represents a trace of messages that can be possibly produced by following a path in the automaton.

Definition 8. *Let $\mathcal{C} = \langle S, s_0, F, M, \delta \rangle$. For states $s, s' \in S$ and a sequence of messages $\overline{m} \in M^*$, the relation $\rightsquigarrow \subseteq S \times M^* \times S$, noted $s \rightsquigarrow_{\overline{m}} s'$ is defined recursively as follows:*

- *for ϵ the empty sequence, $s \rightsquigarrow_\epsilon s'$ if $s = s'$, and $s \not\rightsquigarrow_\epsilon s'$ if $s \neq s'$*
- *if no message m is such that $m \in \overline{m}$ and $\delta(s, m) = s''$ for some $s'' \in S$, then $s \not\rightsquigarrow_{\overline{m}} s'$, otherwise*
- *for every message m and every state $s'' \in S$ such that $m \in \overline{m}$ and $\delta(s, m) = s''$, we have that $s'' \rightsquigarrow_{\overline{m} - m} s'$*

In other words, any scrambling of a trace between s and s' can only be reconstructed in ways that reach s'. We are now ready to define the new outcome function:

Definition 9. *Let $q = s_1, \dots s_k$ and $q' = s'_1, \dots s'_\ell$ be two monitor tokens, and r, r' be their respective desynchronized suffixes. For a possible trace of messages $\overline{m}$ for r and a possible trace of messages $\overline{m}'$ for r', define $\hat{m} = \overline{m} - \overline{m}'$ and $\hat{m}' = \overline{m}' - \overline{m}$. We have $f_2(q, q') = \top$ if and only if there exists a state s such that $s_k \rightsquigarrow_{\hat{m}} s$ and $s'_\ell \rightsquigarrow_{\hat{m}'} s$, $f_2(q, q') = \bot$ if no paths from q and q' can ever reach the same state s, and $f_2(q, q') = \nearrow$ otherwise.*

Informally, the outcome function f_2 checks whether, for two peer's divergent states since they last agreed on the global conversation, there exists a way for both peers to converge back to a common state with the messages they each need to process. If yes, the message is delivered; on the contrary, if the tokens are in two branches of the protocol that can never be reconciled, the function returns $\bot$. Otherwise, the function returns a new value, $\nearrow$ (meaning "wait"), which has for effect of keeping the message in an internal queue for later processing.

For example, in Figure 4 (left), if peer A receives message *b*, stamped with monitor token (1,3), while it is in state

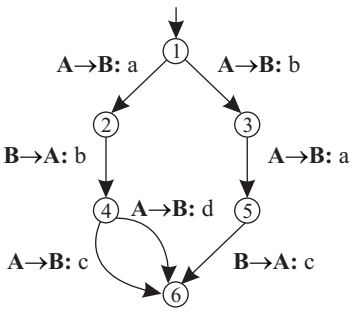

Figure 5. A fixable protocol specification

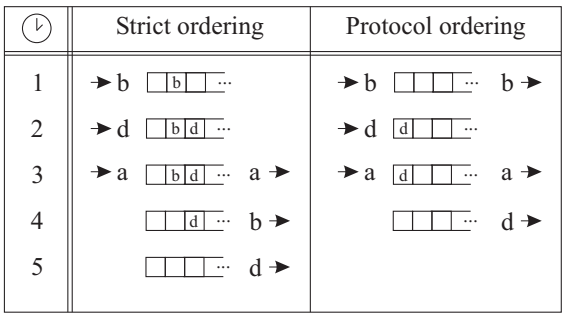

Figure 6. Processing of a scrambled message sequence by messaging components ensuring strict (left) and protocol-compliant (right) ordering.

(1,2), computing the outcome $f_2((1,3),(1,2))$ evaluates that B can process *a* and reach state (1,3,4), while A can process *b* and reach state (1,2,4); both peers are in sync again, and therefore the outcome function returns $\top$. On the contrary, in Figure 4 (right), if peer A receives message *b*, stamped with monitor token (1,3), while it is in state (1,2), computing the outcome $f_2((1,3),(1,2))$ evaluates that A can process *b* and reach state (1,3,4), but B cannot reach state protocol state 4 with *a* as its message to be processed, and therefore the outcome function returns $\perp$, thereby producing the desired behaviour.

Moreover, when a desynchronization is detected between two peers, the MBM automatically forbids any behaviour that will prevent the peers from converging back to a common state. For example, in Figure 5, if A (resp. B) reaches states 2 (resp. 3), the MBM allows it to consume the received message and reach state 4 (resp. 5). Once there, the MBM allows *c* to be sent by A, which will reconcile A and B to state 6; however, the outcome function forbids *d* from being sent, as in B's branch, *d* does not lead to state 6. Hence, the MBM not only detects desynchronizations, it also "repairs" them whenever possible.

B. Consequences on Unreliable Messaging

This new definition of the outcome function has an interesting side effect. By allowing temporary desynchronizations between two peers, the MBM also becomes robust against desynchronizations caused by unreliable messaging. Indeed, f_2 serves messages to the application layer, even if it sends the peers on different paths, but ensures they will reconverge to the same state at some point in the future.

In all related approaches surveyed earlier, no high-level messaging protocol can be specified; therefore, it is impossible to take advantage of the knowledge of the protocol to relax the constraints on the delivery of messages. Yet, Figure 6 shows a situation where such a behaviour would be appropriate. Consider a protocol where A can send to B either the sequence *abd* or the sequence *bad*. Receiving the sequence *bda* is obviously a violation of the protocol; however, the receiver can assume that the sequence was scrambled by the transmission.

Using an MBM, the receiver will be able to relay a protocol-compliant sequence to its application layer, as shown in the right part of Figure 6. Message *b* is first received, and relayed immediately to the application layer; then *d* is received; the outcome function f_2 returns $\nearrow$ and the message is placed in the MBM's waiting queue. When *a* is received, however, it can be relayed, which in turn unlocks *d*. By comparison, the left part of Figure 6 shows how a receiver enforcing a strict ordering of the messages operates on the same scrambled sequence.

In average, each message spends 2 time steps waiting in the queue, compared to only 0.67 time steps with the MBM. The key point is that the definition of the outcome function f_2 allows messages to be delivered *faster* (i.e. with lower queueing time) than traditional messaging such as WS-RM, taking into account the protocol specification to deliver messages using an alternate, yet protocol-compliant, sequence. This can be formalized as follows:

Theorem 2. *Let $\mathcal{C}$ be a protocol specification, and $\overline{m}$ be a sequence of messages, which ends into a state s of the specification. For any permutation $\overline{m}'$ of $\overline{m}$ received by an MBM $\mathcal{R}$, there exists a permutation $\overline{m}''$ such that: $\overline{m}''$ is a path in $\mathcal{C}$, $\overline{m}''$ ends in s, and $\overline{m}''$ is delivered to the application layer.*

Proof: Let $m_1, m_2, \ldots$ be the sequence of messages of $\overline{m}'$. Ensuring that the sequence delivered to the application layer is a sequence of $\mathcal{C}$ is trivial, since $\mathcal{R}$ does not deliver a message unless it can extend the existing conversation with a valid transition. What remains to be shown is that the situation where some messages are "stuck" in the waiting queue at the end of the trace, or where the conversation ends in a different state than s, never happens. Suppose either does. Then there exists a prefix $\overline{m}''_p$ of the sequence of messages delivered to the application layer, leading into some control state s', such that $\overline{m}''_p$ is a path in $\mathcal{C}$, but either no permutation of $\overline{m}' - \overline{m}''_p$ can be completed into a path in $\mathcal{C}$, or some path does not lead to s; in other words, $s' \not\rightsquigarrow_{\overline{m}' - \overline{m}''_p} s$. This contradicts Definition 8, which should have prevented $\mathcal{R}$ from reaching that point. ■

The previous client therefore ensures that any received sequence will be delivered as a protocol-compliant sequence, provided that messages are shuffled but not lost. The speed-up result follows immediately from the fact that the exact-ordering sequences are a subset of the protocol-compliant sequences.

As a side remark, we shall stress that the messenger delivers sequences of messages that are protocol-preserving; they preserve the semantics of a message exchange only as far as the underlying protocol specification does. The case where two sequences of messages *ab* and *ba* are both valid, but *mean* different things, or *perform* different actions and therefore should not be confused, is only partially covered by our approach. In such a case, a sound protocol specification should have the paths *ab* and *ba* reach two distinct states, indicating that two different outcomes result from these sequences.

V. Implementation and Results

To illustrate our point, we implemented the MBM as a middleware tool in Java. The MBMessenger is a simple object that simulates a communication channel. It offers two methods, `put()` and `get()`, to respectively send an XML message to and receive from an arbitrary communication channel.

When the `MBMessenger` is instantiated, a protocol automaton can be specified. Thereafter, when a message is required to be sent through `put()`, the messenger sends it as soon as the protocol allows it, stamping it with its current monitor token. When new messages are requested through `get()`, the messenger empties its communication channel into its waiting multi-set, and then processes each message according to our approach.

A. Methodology

We tested the performance of the MBMessenger by performing a series of experiments. A sender A sends a sequence of messages generated through a random walk in some protocol specification $\mathcal{C}$. To simulate imperfect communication, messages are then randomly shuffled before reaching a receiver B, where they are read one by one. Two cases are then considered. In the first case, A and B communicate through an MBM where protocol monitoring is disabled: the sent messages are stamped with a sequential number, and the received messages are possibly delayed so that they are delivered in the exact order they were sent. This effectively simulates WS-RM's operation when strict ordering is imposed and no messages are lost. In the second case, A and B use the MBM described in this paper: each message is stamped with a monitor token.

We assume that in each discrete time step, at most one message can be added to the messenger, and at most one message can be consumed by the receiver's application layer. In each of these cases, we measured the average waiting time (in time steps) of each message in the messenger's waiting queue, and the maximum size reached by the messenger's waiting queue during the processing of each trace. In order for results to be comparable, the sender in each case generated the same set of traces, and each individual trace was shuffled in the same way in the two scenarios. Therefore, each receiver dealt with the same shuffled sequences of messages, stamped according to either messaging method.

B. Results and Discussion

We then performed these tests, using for $\mathcal{C}$ the protocol for each of two scenarios.

The first one is the Internet Open Trading Protocol (IOTP) [21] which supports commerce on the Internet by providing a familiar trading model and global interoperability. Multiple transactions can be conducted in parallel; however, the IOTP specification lists several dependencies for messages belonging to the same transaction. For example:

1) If the consumer starts the exchange, the first message must be one of either Inquiry, Ping, Authentication
2) The authentication request can only be sent once
3) Once a transaction has been completed, a Cancel message can no longer be sent
4) Two payments sequences cannot overlap, i.e. once a payment request has been issued, the payment must be confirmed before a new payment request be issued

The second scenario is the NetConf protocol [22] which defines a simple mechanism for sending and receiving configuration information for network devices such as routers and switches. NetConf uses XML messages to encapsulate configuration commands and responses and is supported by a wide range of devices. It is possible, for example, to configure a Virtual Private Network (VPN) through the use of XML-PI. Unless otherwise mentioned in Cisco's documentation [23], these commands are atomic and independent of each other. However, the documentation elicits sequential dependencies between some of them:

1) Commands `rd`, `route-target` and `ip vrf forwarding` must be entered in the VRF configuration mode, after `ip vrf`
2) The remaining commands must be entered in the BGP configuration mode, after `router bgp`
3) The commands `route-target` and `ip vrf forwarding` must be applied to a table already created by the command `rd`
4) The command `neighbor update-source` must be applied to an address already included in the routing table with the command `neighbor remote-as`
5) The commands `neighbor activate` and `neighbor send-community` must called after both `neighbor remote-as` and `neighbor update-source`

Table I shows that, for the IOTP scenario, the average waiting time per message is decreased by 52% using MBMs

	Strict ordering	Protocol ordering
Waiting time	1.42	0.69
Queue size	5.58	2.49

NetConf protocol

	Strict ordering	Protocol ordering
Waiting time	0.42	0.38
Queue size	1.81	1.77

IOTP protocol

Table I
AVERAGE WAITING TIME PER MESSAGE AND AVERAGE QUEUE SIZE FOR TWO SCENARIOS.

with protocol ordering. This confirms that providing more flexible conditions on message delivery can reduce the delaying of messages required to preserve an acceptable ordering. Consequently, since messages spend less time in the waiting queue, the average size reached by that queue during an execution is also decreased, by approximately 49%.

However, for the NetConf scenario, waiting time is decreased by 10%, and queue size by 2% only. This can be explained by the fact that in the VPN configuration routine, only two messages can be served first (the two mode commands); as long as either of these message has not been received, the messenger must delay all the others; this does not leave room for the MBMessenger to improve over a strict ordering solution.

In addition, the MBMessenger can be made fully compatible with WS-ReliableMessaging. WS-ReliableMessaging allows for undefined extra elements to be added to a SOAP header. These elements are simply ignored by standard implementations of WS-RM, but MBM-based implementations can take advantage of this additional information to improve over message transmission. This WS-RM "compatibility mode" also accounts for lost messages: these messages can be detected, and retransmissions can be asked, by using the operations provided by WS-RM.

VI. CONCLUSION

In this paper, we have shown how runtime monitors monitoring constraints on the sequence of messages exchanged by two peers can be used to alleviate two issues related to asynchronous messaging. By stamping messages with a suitably defined monitor token, some desynchronizations between communicating peers can be detected before non-compliant messages are exchanged. This has for effect that some unrealizable protocols can become realizable with MBMs. Moreover, MBMs are at the same time protected against some effects of unreliable messaging and can perform reliable messaging that is flexible, i.e. where delivering a protocol-compliant sequence is sufficient, even if its order is not the exact sending sequence. We have shown experimentally that for arbitrarily shuffled message sequences, reliable messaging based on runtime monitoring can decrease delivery time and required queue size compared to strict messaging. These results, however, heavily depend on the underlying protocol. MBMs appear to be best suited for protocols situated midway between those imposing a unique order for their messages, and others where ordering is irrelevant. The results obtained empirically indicate that this principle could be furthered in future work; this includes studying a symbolic representation of the protocol automaton, extending the communication to request-response and multi-party protocols, as well as including "data-aware" protocol constraints.

REFERENCES

[1] H. Foster, S. Uchitel, J. Magee, and J. Kramer, "Model-based analysis of obligations in web service choreography," in *AICT/ICIW*. IEEE Computer Society, 2006, p. 149.

[2] T. Bultan, X. Fu, R. Hull, and J. Su, "Conversation specification: a new approach to design and analysis of e-service composition," in *WWW*, 2003, pp. 403–410.

[3] "IBM conversation support project," 2002, http://www.research.ibm.com/convsupport. [Online]. Available: http://www.research.ibm.com/convsupport

[4] "RosettaNet implementation framework, overview: segments, clusters and PIPs, version v02.06.00," January 2009, http://portal.rosettanet.org.

[5] N. Kavantzas, "Web service choreography description language 1.0," 2004. [Online]. Available: http://www.w3.org/TR/ws-cdl-10/

[6] C. Molina-Jiménez, S. K. Shrivastava, and N. Cook, "Implementing business conversations with consistency guarantees using message-oriented middleware," in *EDOC*. IEEE Computer Society, 2007, pp. 51–62.

[7] J. M. Zaha, M. Dumas, A. ter Hofstede, A. Barros, and G. Decker, "Service interaction modeling: Bridging global and local views," in *EDOC*. IEEE Computer Society, 2006, pp. 45–55.

[8] X. Fu, T. Bultan, and J. Su, "Synchronizability of conversations among web services," *IEEE Trans. Software Eng.*, vol. 31, no. 12, pp. 1042–1055, 2005.

[9] K. P. Birman and T. A. Joseph, "Exploiting virtual synchrony in distributed systems," *SIGOPS Oper. Syst. Rev.*, vol. 21, no. 5, pp. 123–138, 1987.

[10] S. Tai, T. A. Mikalsen, and I. Rouvellou, "Using message-oriented middleware for reliable web services messaging," in *WES*, ser. Lecture Notes in Computer Science, C. Bussler, D. Fensel, M. E. Orlowska, and J. Yang, Eds., vol. 3095. Springer, 2003, pp. 89–104.

[11] R. Bilorusets, D. Box, L. F. Cabrera, D. Davis, D. Ferguson, C. Ferris, T. Freund, M. A. Hondo, J. Ibbotson, L. Jin, C. Kaler, D. Langworthy, A. Lewis, R. Limprecht, S. Lucco, D. Mullen, A. Nadalin, M. Nottingham, D. Orchard, J. Roots, S. Samdarshi, J. Shewchuk, and T. Storey, "Web services reliable messaging protocol (WS-ReliableMessaging)," February 2005. [Online]. Available: http://specs.xmlsoap.org/ws/2005/02/rm/ws-reliablemessaging.pdf

[12] A. Banks, J. Challenger, P. Clarke, D. Davis, R. P. King, K. Witting, A. Donoho, T. Holloway, J. Ibbotson, and S. Todd, "HTTPR specification," Tech. Rep., April 2002. [Online]. Available: http://www.ibm.com/developerworks/library/ws-httprspec/

[13] S. W. Sadiq, M. E. Orlowska, W. Sadiq, and K. A. Schulz, "Facilitating business process management with harmonized messaging," in *ICEIS (1)*, 2004, pp. 30–36.

[14] P. Huifang, Z. Xingshe, Y. Zhiyi, and G. Jianhua, "A flexible hybrid communication model based messaging middleware," in *ISADS*. IEEE Computer Society, April 2005, pp. 289–294.

[15] A. Erradi and P. Maheshwari, "wsBus: QoS-aware middleware for reliable web services interactions," in *EEE*. IEEE Computer Society, 2005, pp. 634–639.

[16] P. Maheshwari, H. Tang, and R. Liang, "Enhancing web services with message-oriented middleware," in *ICWS*. IEEE Computer Society, 2004, pp. 524–531.

[17] S. Parkin, D. Ingham, and G. Morgan, "A message oriented middleware solution enabling non-repudiation evidence generation for reliable web services," in *ISAS*, ser. Lecture Notes in Computer Science, M. Malek, M. Reitenspieß, and A. P. A. van Moorsel, Eds., vol. 4526. Springer, 2007, pp. 9–19.

[18] A. Charfi, B. Schmeling, and M. Mezini, "Reliable messaging for BPEL processes," in *ICWS*. IEEE Computer Society, 2006, pp. 293–302.

[19] S. Hallé and R. Villemaire, "Runtime monitoring of message-based workflows with data," in *EDOC*. IEEE Computer Society, 2008, pp. 63–72.

[20] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[21] D. Burdett, "Internet open trading protocol (IOTP)," August 2000. [Online]. Available: http://www.ietf.org/rfc/rfc2801.txt

[22] R. Enns, "Netconf configuration protocol, IETF Internet draft," p. 103, February 2006. [Online]. Available: http://www.ietf.org/internet-drafts/draft-ietf-netconf-prot-12.txt

[23] "Configuring a basic MPLS VPN, Cisco systems document 13733," Cisco Systems, Tech. Rep., 2005.