Data Sharing vs. Message Passing: Synergy or Incompatibility? An Implementation-Driven Case Study

Matteo Ceriotti[1], Amy L. Murphy[2], and Gian Pietro Picco[3]

[1]FBK-IRST, Povo, Italy, & University of Trento, Italy, ceriotti@itc.it
[2]FBK-IRST, Povo, Italy, & University of Lugano, Switzerland, murphy@itc.it
[3]University of Trento, Italy, picco@dit.unitn.it

ABSTRACT

Coordination models can be roughly categorized into *data sharing* and *message passing*, based on whether the information necessary to coordination is persistently stored and shared, or instead is only transiently available during communication. Generally speaking, approaches based on data sharing are more expressive and provide full decoupling in space and time. The alternative approach requires the simultaneous presence of the coordinated parties, but are typically more scalable. Prominent examples are, respectively, tuple spaces and publish-subscribe.

An open research question is whether it is possible to exploit in synergy the best of these two approaches, e.g. by implementing the more complex data sharing coordination on top of the more lightweight message passing one. In this paper, we seek an answer to this question in a pragmatic way: we analyze an implementation of the LIME tuple space middleware on top of REDS, an open source publish-subscribe system. Our implementation-driven style of investigation forces us to face details that do not surface when reasoning in the abstract about the nature and expressiveness of the models. We report about lessons we learned in this experience, and propose an extension to the publish-subscribe model that, albeit useful per se, constitutes a more effective foundation for data sharing coordination models.

Keywords

Tuple spaces, publish-subscribe, middleware

1. INTRODUCTION

Coordination involves the exchange of information that is used by processes to govern their own actions and affect each other. The interactions among the coordinating parties can be driven by the actual values of shared data or by message passing. This distinction clearly identifies two categories for coordination models, as recognized in [13] where several examples are also provided. In the first *data sharing* models, often inspired by the tuple space concept originally introduced by Linda [9], coordination occurs by manipulating the current *state* of the system, as made globally available by the coordination infrastructure. For instance, Linda enables one to read (or withdraw) one among the tuples hosted by the tuple space, or change its content by adding a new tuple. In this paper, we refer to this kind of coordination built on data sharing also as *stateful*, precisely because it relies on a persistent, globally available representation of the information necessary to coordinate processes. The second alternative coordination models, based on *message passing*, are inherently *stateless*, such as event-based systems [11]. Indeed, in these models coordination is achieved by relying only on the information contained in a message transiently available to the coordinated parties, without any globally-accessible state information. For instance, in publish-subscribe a message published by a process is received only by those subscribers who are interested in the message content or topic. The coordination information is entirely contained in the message, is accessible only during the message exchange, and is discarded by the coordination layer after delivery to the application process. Note that here we define the notion of state by looking only at the coordination model: the coordination *system* implementing such model will of course contain some state (e.g., the subscriptions issued by the various components for filtering messages of interest).

The two approaches present clear differences. Data sharing models provide full decoupling in space and time, enabling exchange of information even when processes are not active at the same time. In a distributed setting this typically comes at the price of reduced scalability, due to the need to enforce some degree of consistency of the coordination state across the distributed system. On the other hand, message passing models are more easily and efficiently implementable, in that they foster interactions that are temporally and spatially confined to the coordinated parties involved in the message exchange.

Often, the two models are considered alternatives that, as mentioned above, provide a different balance among various tradeoffs. Some researchers have tried to reconcile the two approaches by *extending* stateless models and providing some degree of support for state-based operations. For instance, in the context of the publish-subscribe model, the subscription language of the PADRES [8] system is augmented with the possibility to operate on the history of the events published in the system; the work in [6] proposes a query language to express complex subscriptions over published events. In [1] the data shared for the coordination is composed of both the published events that are persistently stored inside the publish-subscribe middleware and the active subscriptions; the primitives are extended with the possibility to query such data and remove the published events from the shared state.

In this paper, instead, we take a different perspective motivated by the following questions: *Can we implement a form of stateful, data sharing coordination entirely on top of a stateless, message*

SAC'08 March 16-20, 2008, Fortaleza, Ceará, Brazil

passing one? If so, to what extent the advantages of the underlying stateless approach can be exploited by the stateful one?

A restricted form of this question was previously posed and tackled from a strictly theoretical perspective in [2], where the authors present a mapping of the original Linda model onto the publish-subscribe model. Nevertheless, their approach both abuses the stateless nature of publish-subscribe by using subscription filters to persistently store application state and, although technically correct, its possible implementation raises many practical concerns. Instead, we chose a pragmatic approach, using an implementation-driven *case study* as a concrete example whose realization yields practical insights into system issues. Specifically, we chose as a representative of the class of data sharing coordination models LIME [12], and publish-subscribe as a representative of the ones based on message passing. The case study consists of investigating if and how the LIME system can be (re)implemented on top of the REDS [5], a publish-subscribe system that, by being open-source and extensible, provides a good foundation for our experiment. The two models, and related systems, are briefly described in Section 2.

Section 3 presents the system architecture considered throughout this paper. Section 4 reports about a first integration of the two approaches, where the publish-subscribe support provided by REDS is used unmodified by the LIME implementation. This strategy has a number of drawbacks caused, as one could expect, mostly by the semantic gap between the two layers. We identify the ability to *collect* state information as the fundamental building block missing from the publish-subscribe layer. Therefore, in Section 5 we propose *publish-subscribe-collect,* an extension of publish-subscribe that, by offering this capability, provides a richer foundation for the development of data-driven coordination models. An enhanced implementation of LIME based on publish-subscribe-collect is analyzed in Section 6 and the inner workings of the collect primitive are outlined in Section 7. We analyze the results of our case study in Section 8 and provide concluding remarks in Section 9.

The reader should be warned, however, that we are not claiming here that the scalability issues and other idiosyncrasies of stateful approaches can magically vanish. Even with our enhanced publish-subscribe-collect model, there are situations where the efficiency drawbacks of stateful approaches surface. These are indeed intrinsic in data sharing models, and are the main problem grieving the middleware designer implementing them. In summary, in this paper we make the two following contributions:

- We provide a careful, *implementation-driven* analysis of the opportunities and pitfalls in integrating two popular coordination models. To the best of our knowledge, no other similar analysis exists in the literature.
- We introduce a novel *extension* of the publish-subscribe model, which can be useful per se or as a building block for higher-level data sharing coordination abstractions.

2. BACKGROUND

Before describing the integration of the data sharing and message passing models, we first discuss them in isolation, introducing the systems we selected for the case study.

2.1 Transiently Shared Tuple Spaces

The tuple space model, made popular by Linda [9], supports the coordination of processes by providing a shared medium accessed through a minimum set of operations, namely insertion (**out**), reading (**rd**) and removal (**in**). The actual information is described by tuples, a sequence of typed parameters, such as ⟨"foo", 9, 29.4⟩ and the access to them is performed using templates composed by actual (i.e. values) and formal (i.e. "wild cards") fields. The resulting interaction is decoupled in time and space and the coordination is based on the content of the available tuples.

The model resonates with the distributed mobile setting as demonstrated by LIME [12], which breaks up the tuple space into many tuple spaces each permanently associated to a single node and defines rules for transient sharing of the individual tuple spaces based on connectivity. The scope of the sharing can be restricted by assigning a tuple space a name, and allowing each node to own more than one named tuple space. The abstraction provided to the application is that of a local tuple space containing tuples coming from all the units currently accessible. This *transiently shared tuple space* can be accessed through a set of operations enlarged with respect to those of Linda, namely probing operations (i.e. **rdp** and **inp**) that do not block the requester if a result is not available and bulk operations (i.e. **rdg** and **ing**) that return a set of matching tuples.

LIME also extends the model with a notion of *reaction*; it is defined as a listener active over the tuple space for all (or just one) of the tuples matching the associated template. The listener fires both with the tuples already available in the system at the moment the reaction is submitted and those inserted later. This behavior highlights the dependency of the middleware on both the state of the system at the moment the operation is issued and the following changes to it.

2.2 Publish-subscribe

The publish-subscribe model [7] supports anonymous, multi-point communication among publishers and subscribers. Subscribers specify their interest in messages through a subscription containing a message *filter*. Filters can be based on a message subject or content. Our work focuses on the latter, where subscription filters are predicates over the content of the published messages. When a publisher sends a message that matches a previously installed subscription, the message is forwarded to all the corresponding subscriber(s).

The matching of messages against subscriptions, as well as the consequent message forwarding, are the responsibility by the *message dispatcher*, which effectively decouples publishers and subscribers. In terms of implementation, a message dispatcher is easily provided by a dedicated server, leading to the usual scalability and single-point-of-failure problems of centralized architectures. Therefore, in this work we focus on distributed implementations of the message dispatcher, where the dispatching functionality is provided by application-level routers called *brokers*. In particular, in our case study, we used the REDS [5] system, a distributed publish-subscribe framework whose reconfigurable and extensible architecture allows us to easily exploit a standard publish-subscribe system as well as to extend it to encompass our own extensions, discussed in Section 5.

3. SYSTEM ARCHITECTURE

Figure 1 depicts the architecture of a single host, decomposed into three distinct layers: the application, the LIME middleware, and the publish-subscribe middleware. Conceptually the top two layers focus on data sharing coordination while the lowest is message passing. The system is composed of multiple hosts, and all communication is managed by the lowest layer.

The LIME application is composed of multiple, distributed agents interacting with one another through the interface to the transiently shared tuple space (ITS). All communication remains hidden beneath the tuple space interface. Our work builds on a recent version of the LIME middleware that decouples the underlying transport layer from the local tuple space management. Its goal is to

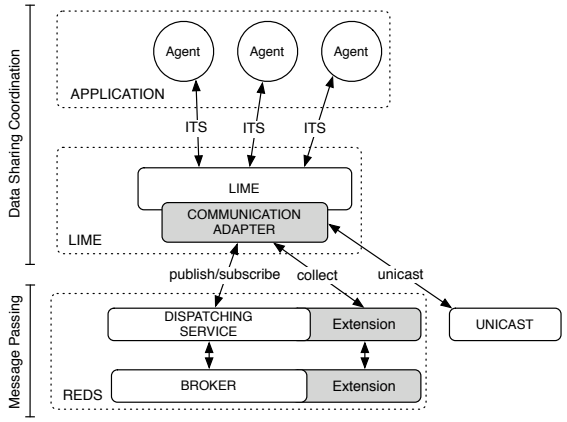

Figure 1: Components on a single host. Grey components are those affected by our case study.

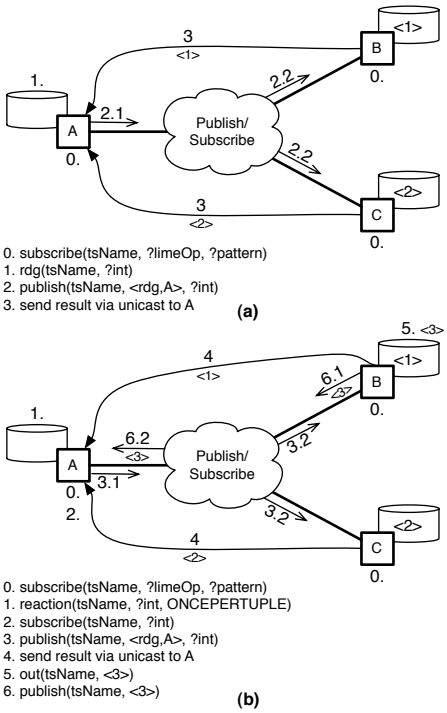

Figure 2: Mapping LIME operations onto the unicast and publish-subscribe primitives: (a) **rdg** and (b) ONCEPERTUPLE reaction. We assume all hosts to own one tuple space with the same name (tsName).

allow LIME to run on top of any transport layer for which a communication adapter conforming to a standard interface has been provided. Therefore, LIME is internally divided into a component managing the local tuple space and a communication adapter, controlling all interactions with other nodes in the system. Essentially this adapter converts locally-issued LIME operations involving remote hosts into a sequence of communication actions. In a similar way, when an operation issued remotely is received, a listener is in charge of accessing the local tuple space and replying properly.

In this paper we describe the actions taken by the communication adapter to work either with publish-subscribe (Section 4) or publish-subscribe-collect (Section 6). Both implementations are instantiations of the REDS framework. To make each host as independent as possible, we opted to instantiate both the dispatcher and broker on all hosts, using local communication between LIME and the dispatcher, and between the dispatcher and the broker.

While in principle all operations can be carried out with either the publish-subscribe or the publish-subscribe-collect primitives, our case study revealed that some operations are best managed with one-to-one communication between hosts, motivating the inclusion of a specific module to handle unicast interactions.

4. LIME OVER PUB/SUB

We begin our investigation with a study of how the federated tuple space model of LIME can be implemented on top of the traditional publish-subscribe model. Every LIME operation involving multiple hosts is actually a sequence of actions executed by the communication adapter. These actions must be mapped onto either unicast or publish-subscribe interfaces. This section outlines these mappings for operations issued over the whole federated tuple space. For operations issued locally, or over a specific, remote tuple space, the unicast component is used exclusively. This section is organized along the different kinds of LIME operations, namely, non-blocking, reactions, and blocking operations.

Non-Blocking Operations. **rdg**, one of the non-blocking operations provided by LIME, returns to the operation initiator all the existing tuples matching a specified pattern, or **null** if no matching tuple exists. In Figure 2(a), we show an example of the interactions required to achieve these semantics over publish-subscribe. We employ the query/advertise paradigm [10] in which each node subscribes (advertises) to receive published messages (queries) it can respond to. The idea for LIME is for each host to *subscribe* (step 0 in the Figure) to receive non-blocking operations (**rdp**, **inp**, **rdg**, **ing**) destined for a named tuple space held by that node. When a non-blocking operation is issued (step 1), it is *published* (step 2), along with the identity of the initiator. The publish-subscribe infrastructure receives the message (2.1 in the Figure), delivers it to each of the subscribers (2.2 in the Figure), i.e. owners of the queried tuple space, then these nodes look locally for matching tuples. If a match is found, a copy of each matching tuple is sent via unicast (step 3) back to the query initiator. After a given timeout, all received results are returned to the agent; if no response has been received, we return **null** even if some matching tuple exists but its availability is notified after the expiration of the timeout.

All other non-blocking operations follow a similar implementation. For a **rdp**, the first result is returned to the agent, while the others are discarded. **inp** and **ing** follow the corresponding read operation and append further actions to remove the matching tuple(s). Specifically, an additional, direct (unicast) interaction is issued to remove the tuple(s) from the hosting node(s).

Reactions. LIME offers the ability to execute a user defined listener either once for one single matching tuple or once for every tuple available in the system. The available primitives that meet such requirements are respectively ONCE and ONCEPERTUPLE reactions. The tuples that satisfy these operations are both the ones already available in the system at the time of installation as well as those inserted in the shared tuple space later. Achieving the presented semantics requires a copy of the matching tuples to be sent to the reaction initiator where the listener can be executed.

For ONCEPERTUPLE reactions, shown in Figure 2(b), this is realized over publish-subscribe with a combination of two operations,

namely a **rdg** (steps 0, 3 and 4) to retrieve all existing matching tuples, and a subscription (step 2) for matching tuples later inserted into the tuple space. The need for the former is due to the semantics of the subscription that operates only on the data made available by publication after the activation but not on the information already present in the state of the system when the subscription is issued. We further require that all nodes publish all tuples as they are inserted (step 6), allowing tuples to flow to all hosts with registered reactions (6.1 and 6.2 in the Figure).

Since ONCE reactions require only a single result, we substitute the **rdg** with a **rdp**. Nevertheless, if the result is **null**, a subscription is issued and later removed by an unsubscription as soon as a result, different from **null**, is returned.

Blocking Operations. The blocking operations, that return one matching tuple, are nicely built on top of the previously described ONCE reactions. A listener can be defined that blocks the issuing agent until a result is found and then the lock is released and a copy of the tuple, in the case of **rd**, returned. For an **in**, an additional out-of-band, unicast communication removes the tuple. After its successful removal, the unsubscription is executed.

Discussion. At first sight, LIME and publish-subscribe have some key points in common, which suggest the potential for synergy between the two models. They both support decoupling between data producers and consumers, which interact without using node identifiers, instead using pattern matching over relevant data as the main mechanism for establishing coordination among the parties. Moreover, both models provide a form of asynchronous notification. In publish-subscribe, the very model revolves around the concept that a process is notified whenever a message matching one of its subscription filters is published by some other component in the network. In LIME, reactions tie a similar notification functionality to state changes: the programmer can essentially "subscribe" to changes in the tuple space, which are implicitly "published" in the transiently shared tuple space spanning all components.

On the other hand, an analysis of our implementation reveals that, in practice, the two models also present some incompatibility, whose source can be traced back to the state dimension present in LIME and absent in publish-subscribe. In the latter, message passing limits the scope of interactions uniquely to the content of the message itself. Instead, in LIME and in general in data sharing models, operations potentially span the whole distributed state. Performing operations in these models involves two steps: *state collection* and *state update*.

Consider a blocking **rd** operation. First, the system must be scanned to see if a matching tuple exists—state collection. If this is not the case, the system must be monitored to make sure that if and when a matching tuple appears, the proper processing occurs—state update. Clearly, the publish-subscribe model is inherently well-suited for providing support for state update, and we exploit this ability in the mechanisms we just illustrated. Unfortunately, however, its application to state collection is somewhat cumbersome. Indeed, the publish-subscribe layer is oblivious of the message content, and simply disseminates its messages (including those containing operation requests) to all interested parties—each time this is requested, regardless of the previous interactions. Instead, when applied to state collection, one would like to take into account nodes that, when last queried, did not have any result and have not communicated a state update since then. Sending operations to these nodes is wasteful, the more so the bigger the scale of the system. These kinds of optimizations, however, are clearly not possible without fundamentally changing not only the publish-subscribe implementation, but also the very nature of the publish-subscribe model.

Similarly, the cooperation of the state collection and update phases is also problematic. In our solution, to remain faithful to our goal of reusing the publish-subscribe system *as is*, we significantly increased the complexity of the communication adapter component in Figure 1. Indeed, this component must perform significant bookkeeping to reconcile results coming through separate channels—the unicast messages carrying query results and the published message notifications coming as a result of a subscription installation and the subsequent appearance of a matching tuple.

Therefore, the synergies between the LIME and publish-subscribe models are undermined by the mismatch between the two in terms of their focus on state. Such limitations can be overcome with ad-hoc solutions, as we did in the design of our communication adapter. However, this clearly leads to non-reusable solutions. Instead, based on our observations, we propose an extension of the publish-subscribe model that, without changing its fundamental nature, provides just what is needed in terms of expressiveness to effectively deal with state collection: a *collect* primitive that reunites the two perspectives above and can be implemented in a way optimized for dealing with state.

5. THE COLLECT PRIMITIVE

Based on the considerations above, we propose publish-subscribe-collect, an extension to publish-subscribe that explicitly provides access to application state. While our motivation was to provide better support for LIME, the resulting model is general enough to support any data sharing coordination. This section introduces the model and its primitives, while the following sections return to its effectiveness for supporting LIME as well as implementation concerns.

5.1 Enabling State Access

Our model is based on three concepts that address *application* state: local state, state patterns, and state updates. The definitions of all three are left to the application, and effectively enable a sharp separation of concerns between the application and the coordination middleware, leading to a flexible model that enjoys wide applicability.

The *local state* of an application is whatever the latter deems relevant for coordination purposes. More interesting is the notion of *state pattern*, which describes the (remote) state an application is interested in. These patterns are analogous to the event patterns of publish-subscribe, but are defined and evaluated by the application rather than the middleware. When the middleware receives a pattern as a parameter of the *collect* operation, it routes it to the distributed hosts, where it is passed up to the application for evaluation. The pattern is therefore evaluated over the local state of the receiving node only, not over a combination of states on multiple hosts. Further, the response to the query is completely defined by the application, but can include a special response, *empty*. Again, the precise definition of *empty* is left to the application. For example, it may be used to report that the application is in an initial state, or that it has no state matching the supplied pattern. In any case, *empty* has a special meaning for certain variations of *collect*, as detailed in the next section.

Additionally, applications publish *state update* messages, describing changes to their local state. The intuition is that an application, through *collect*, can receive continuous updates about a remote node's changing state, similar to subscriptions to receive published messages in the standard publish-subscribe model. Identifying which updates to forward to which hosts is a matter of matching

Table 1: The six variants of collect.

	ANY	**ALL**
PROBING	If all nodes *empty* and no matching state update found, returns *empty*. Otherwise returns one, non-empty state or matching state update.	Queries the state of all nodes. Returns *empty* or non-empty for each, along with the matching state updates encountered during the evaluation.
SINGLE	Returns one, non-empty state or one matching state update. If all nodes *empty*, remains installed until one node has a non-empty state or publishes a matching state update.	Returns one, non-empty state for each node in the system. If any node is *empty*, remains installed until that node has a non-empty local state or publishes a matching state update.
PERSISTENT	Non-deterministically selects one node with non-empty state: returns this state and all future matching state updates.	Returns the state of all nodes, along with all future matching state updates.

a state pattern against the update messages. It is interesting to note that, in principle, matching a pattern to an update message may also be useful for providing a response to a state query. However, because the contents of the state update are entirely defined by the application, such a match may or may not make sense. For instance, consider the case where the notion of local state is represented by HTML pages, and a state update simply contains a *diff* between an old and a new version of the same page. In this case, it is unlikely that the message carrying the state update (the diff) satisfies the query, that instead is likely to be formulated in terms of the relevant state (the whole page). Nevertheless, other applications may provide sufficient state information in a state update message: this is the case of LIME, where the state update is simply the new tuple that appeared in the tuple space. Ultimately, since the content of the state update messages and the definition of state patterns are in the hands of the application programmer, it is sufficient to encode in their matching rules if and how a state update matches a state pattern.

In summary, the application manages its own local state, publishes messages describing state changes, issues queries containing patterns describing the desired state (matching either local state data or published state updates), and responds to queries with local matching state information.

5.2 The Publish-Subscribe-Collect Model

The primary mechanism for a publish-subscribe-collect application to access state is through a new operation, *collect*. This allows it to query the current local state of the nodes in the system for matching, as well as to monitor changes to such state. The collect primitive is parametrized along two dimensions: space and time.

For the former, we consider the extent of the operation in terms of the hosts involved in the query. This can vary from a single node to all those available in the system. While theoretically any scope constraint may be meaningful, for practical reasons we focus only on the extremes, providing operations over *all* hosts participating in the system or to *any* single host. To maintain anonymous communication among participants, the *any* option non-deterministically selects a suitable participant.

To define the time dimension, we consider the cause of the operation termination: implicit after the operation is satisfied, or explicit through cancellation by the application. Further, we recognize two cases for satisfying an operation, namely accepting an empty state or requiring a non-empty state response. This yields three options for defining the operation duration, namely *probing* that accepts an empty state, *single* that is satisfied with a non-empty state response, and *persistent* that remains active until explicitly cancelled by the application.

The combinations of the space and time dimensions yield six variations of the collect operation, outlined in Table 1. The semantics of publish and subscribe are unaltered.

6. LIME OVER PUB/SUB/COLLECT

In comparison to the mapping of LIME over publish-subscribe presented in Section 4, the mapping over publish-subscribe-collect is much more straightforward. The system architecture remains that of Figure 1, with the use of the *collect* extensions inside REDS. The unicast component is retained for use with both LIME operations issued to specific remote hosts as well as for tuple removal operations (e.g., **in**, **inp**, **ing**).

As before, adapting LIME requires instantiation of the communication adapter to respond to queries and to map LIME operation actions to publish-subscribe-collect operations. Unlike our original adapter described in Section 4 where the publish-subscribe layer is state-agnostic, publish-subscribe-collect explicitly manages both state queries and their responses. Therefore, when a collect query arrives at a host, a listener residing in the communication adapter is called and the local tuple space is queried; the result is then returned to the publish-subscribe-collect layer. This is notably different from our previous mapping, in which the communication adapter explicitly routed replies back to the operation initiator. When a query response arrives at a host, a second listener redirects the response to the pending LIME operation, which then forwards it to the application agent. Again, this is cleaner than our previous solution as all replies arrive through the middleware instead of a combination of the middleware and the unicast module.

As expected, the tuple space forms the state to which publish-subscribe-collect provides access. Queries contain tuple patterns, and indicate how many tuples are required in the response (e.g., one for **rdp**, all matching for **rdg**). State update messages are used by LIME to announce the insertion of a tuple, and therefore contain a copy of the inserted tuple. When a single tuple is sufficient for a query, the pattern allows matching to state update messages.

This section details the mapping of LIME onto publish-subscribe-collect, while Table 2 provides a brief overview.

Table 2: Mapping LIME to publish-subscribe-collect.

	ANY	ALL
PROBING	**rdp**	**rdg**
SINGLE	**rd**, **reactTo** (ONCE)	
PERSISTENT		**reactTo** (ONCEPERTUPLE)

Non-Blocking Operations. The non-blocking operations, i.e. **rdg**, **rdp**, **ing**, **inp**, provide results about the current tuples in the system, precisely matching the semantics of *probing-collect*. Specifically, a **rdg** that requires state from every available host uses the *all* variation with a pattern requesting all matching tuples be returned. Instead, **rdp** maps to the *any* variation to receive only a single result. Its pattern requires only a single result, and allows matching to a state update message. **ing** and **inp** are implemented

as a combination of a read operation followed by a unicast to remove the tuple(s).

Reactions. ONCEPERTUPLE reactions, which combine state collection and subscription, map precisely to *persistent-collect-all*. Their pattern selects all matching tuples and matches state update messages carrying matching tuples. In comparison to our previous solution that used a decoupled combination of a **rdg** and a subscription, this solution achieves the same semantics with a single operation. Finally, a ONCE reaction requires only a single matching tuple, hence it is mapped to a *single-collect-any* operation.

Blocking Operations. As noted previously, **rd** has the same semantics as a ONCE reaction over the federated tuple space. Therefore, it is also mapped to a *single-collect-any*. To implement an **in**, the communication adapter first uses *single-collect-any* followed by a unicast interaction to perform the tuple removal. If the tuple is concurrently removed by another process, the entire procedure is repeated until a result is successfully obtained.

Discussion. As is evident from the brevity of the description, the mapping of LIME operations to publish-subscribe-collect is straightforward, demonstrating the ability of our new model to elegantly support the stateful coordination of LIME. The use of the different state patterns to control the number of tuples contained in a response exemplifies the flexibility of the approach. Further, the details of the query distribution and response routing remain completely hidden to the application, emphasizing the ability of the middleware to maintain the separation of concerns between the application and the coordination layers.

7. ON IMPLEMENTING COLLECT

Having outlined the publish-subscribe-collect model and its applicability for LIME, we now explore implementation issues made evident during prototyping.[1] Our prototype is an instantiation of the REDS framework that adds the collect operation on top of an implementation of the subscription forwarding routing strategy [3]. Before moving into the details of implementing and optimizing collect, we first provide a brief introduction to subscription forwarding, as it influences our implementation.

Subscription Forwarding. Subscription forwarding is one of the most popular publish-subscribe routing strategies. It assumes distributed brokers are arranged in a unrooted tree topology. Subscriptions propagate along these tree branches from the subscriber to all nodes of the system establishing routes for matching messages to follow from each node back toward the subscriber. To avoid propagating all subscriptions throughout the entire tree, a common optimization stops forwarding subscriptions for any pattern that has already been subscribed to. The intuition is that the existing routes are sufficient to properly route messages for the new subscription.

Regarding efficiency, subscription forwarding assumes the number of messages is much higher than the number of subscriptions. Hence, the overhead to distribute subscriptions to all nodes in the tree is compensated by the efficient routes followed by messages.

7.1 Probing-Collect

The *probing* versions of *collect* terminate implicitly, possibly with *empty* as a result. Additionally, matching state update messages can be part of the response. We begin our discussion with the *all* variant, as it is the most general.

Taking inspiration from subscription forwarding, the *probing-collect* operation propagates from the initiator to all nodes in the system. When it reaches a leaf node, the state is queried and the response forwarded to its parent, an intermediate node on the path to the initiator. Each intermediate node collects the state from its children, combines the result with its own state and any matching state update messages it has processed while waiting for its responses from its children, and forwards the combined result to its parent. Eventually the initiator receives state information from each of its children and returns the result to the waiting application. Importantly, no timeouts are required to identify when the full state has been collected.

Implementing the *any* variant requires a slight modification of this form of state collection over a tree. Because only a single result is required, if a non-empty state is encountered during query propagation, this state is returned, the propagation halted, and the collection tree removed. Alternately, if a node waiting for responses from its children processes a matching state update message, this is returned to the initiator and the query cancelled. If the query result is *empty*, the processing for both *any* and *all* are identical.

It is worth noting that the *all* variant essentially returns the global, distributed system state. Although each state is kept locally consistent by the application, the combination of multiple local states raises the concern about the consistency. Comparison of our implementation of the state collection to the original Chandy-Lamport distributed snapshot [4] reveals that the processing is identical, collecting local states and the state update messages. Nevertheless, because the definition of these elements is in the hands of the application, they may or may not be sufficient to reconstruct a consistent state. To improve the efficiency of the snapshot implementation, we consider that multiple nodes may issue identical operations simultaneously. Rather than run separate snapshots, partial results such as the states along common tree branches can be shared. We therefore exploit the Kearns and Spezialetti distributed snapshot [14], which efficiently supports concurrent initiators.

7.2 Single-Collect

The *single* variants of *collect* require state from one or all hosts. The *all* variation propagates throughout the system along the tree, establishing routes to the initiator, similar to the *probing-collect-all* propagation described above. When a node receives the query, if it has a non-empty state, this is forwarded immediately to the initiator. Otherwise, the middleware monitors the host for locally published state update messages that match the query. Unfortunately, if the application does not allow state update messages to match the query, then the query may never be satisfied. Therefore, after receiving a local state update message, the middleware directly queries the local node for its matching state. The intuition is that even if a state update message does not match the pending query, it does signify that the local state has changed, and therefore a new request may return a non-empty state. When the node and all its children have reported non-empty states, the route to the initiator is no longer necessary and is removed.

When only a single response is required (*single-collect-any*), propagation halts as soon as a node with a non-empty state is encountered. However, unlike *probing-collect-any* which collects *empty* responses, *single-collect-any* remains active until one nodes reports a non-empty state, either through a state update message or a local query by the middleware. In either case, the result is returned to the initiator and the operation cancelled.

[1] Prototype URL suppressed for blind review.

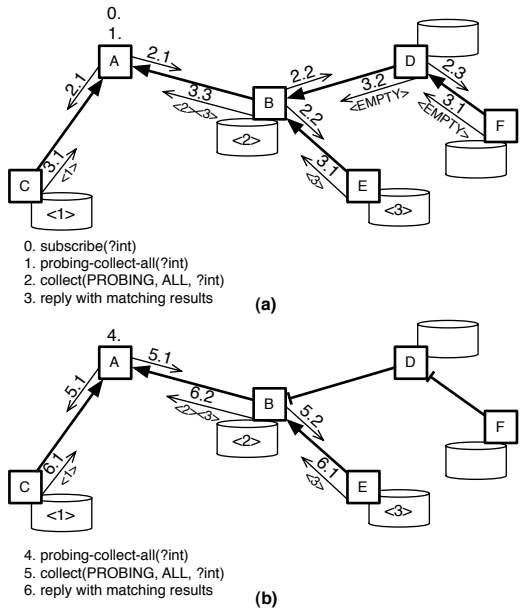

Figure 3: Arrows between nodes indicate routes for messages toward subscribers. (a) Without pruning, all *probing-collect-all* operations reach all hosts. (b) With pruning, subsequent *probing-collect-all* operations are limited to parts of the network with state. Nodes such as D whose descendants previously reported *empty* and have not since sent state changes (indicated by the $\top$ on the links) are eliminated from the request.

7.3 Persistent-Collect

The *persistent* variants of *collect* bridge the gap between subscription to future state changes and collection of current state. As such, the implementation is an integration of the state collection described previously for the *probing* variations and a concurrent subscription. Specifically, as an *all* version of the *persistent-collect* propagates, the local state query is performed as usual, and a subscription is installed to forward matching state updates. However, unlike traditional subscription installation which stops propagating when an identical subscription is encountered, the query must continue to propagate in order to collect the state of all nodes. On the other hand, for efficiency reasons, the subscriptions are not duplicated in the subscription tables.

The *any* version of *persistent-collect* non-deterministically *selects* a host, collects its state, and monitors its updates. This selection requires establishing only a single path from a node with non-empty state and the operation initiator. To identify which path to establish, we use the previously described mechanism to disseminate the query along the tree. When a node with a non-empty state is encountered, the operation stops propagating, the state is returned, and all other routes are removed. If multiple hosts from different branches of the tree reply, the initiator non-deterministically selects one and removes the routes from the others.

7.4 Optimizing State Retrieval

Considering the operation cost, our implementation of the *any* variants of *collect* represent a significant improvement over the corresponding implementation using traditional publish-subscribe. For example, consider that a single state collection, such as **rdp**, implemented on top of publish-subscribe publishes a query which is routed to all subscribed hosts (in LIME, those with the same named tuple space). Although only a single response is required, all hosts receive the query and all those with matching tuples. Instead, with the implementation described here, the query ceases to propagate when a matching state is found, limiting the scope of the operation. Although our propagation mechanism simultaneously spreads the query to all neighbors, and therefore potentially multiple hosts may respond, the savings can be significant especially in large systems.

On the other hand, the overhead for *collect* operations that retrieve state from all hosts remains a concern. Returning to our initial motivation from LIME, we considered that significant savings could be achieved if we cache tuples from previous queries. For example, in Figure 3, if node B caches the returned tuple from E, subsequent queries can return the cached value without additional messages along E's sub-tree. However, because tuple removals are not published as state update messages, keeping these caches up to date is impossible.

Nevertheless, it is possible to cache if a branch returned *empty*. Consider the example in Figure 3(a) in which A issues a *probing-collect-all* operation (step 1). All nodes receive the forwarded request (step 2) and reply (step 3). Without optimizations, subsequent *probing-collect-all* queries have the same behavior, again reaching all tree nodes, including the empty branch of the tree containing D and F. However, consider the case where B has an existing subscription (step 0). This means that state updates (e.g., inserted tuples) at D and F will be forwarded through B. Therefore, it is safe for B to cache the *empty* result from D, because this *empty* cache status is invalidated by any state update arriving from D.

Implementing this is straightforward, as it only requires monitoring *empty* queries for patterns where an identical subscription already exists. The annotation is visualized in the figure with a $\top$ on the links from F to D and D to B. It is worth noting that this optimization applies to all collect variants, and although it requires an existing subscription to allow subsequent pruning, we expect this to be common. For example in LIME, the installation of a reaction is sufficient to establish the subscription (through the *persistent-collect-all*), after which all subsequent queries can benefit from pruning.

8. DISCUSSION

One of our initial goals was to identify the synergies and incompatibilities between two coordination styles through a case study. In the end, we uncovered both. On the positive side, it is clear that the data sharing model of LIME can be reasonably implemented on top of a stateless message passing model, namely publish-subscribe-collect. The introduction of state-aware operations into the corresponding middleware simplifies the implementation of stateful constructs, while knowledge of the operation semantics opens up opportunities for overhead reduction. Interestingly, this is all achieved through a model that maintains the separation of concerns between data distribution and data management, yielding a very flexible middleware useful for any data sharing application, not only LIME.

On the other hand, providing access to the entire system state presents inherent scalability problems. While the optimizations presented in Section 7.4 do reduce the overhead, they are applicable only in conditions where entire branches of the system report the *empty* state. We intend to explore additional optimizations such as altering the topology to encourage branches with *empty* state or even caching state on stable branches. In general, we intend to pursue a quantitative evaluation of our publish-subscribe-collect implementation in order to identify the best avenues for optimization.

Finally, it is worth noting that our investigation started with LIME, a model intended for mobile ad hoc networks (MANET). A version

of REDS has been implemented for MANET and, as a natural consequence, we have begun to investigate if the collect operation can be added within the mobility constraints to yield a version of publish-subscribe-collect for MANET. Our initial investigations reveal that the mapping is feasible, however the semantics of the operations over *all* nodes must be redefined and additional state and processing must be added to each node when a tree branch is added or removed during the processing of a query. Nevertheless, we believe that with minor modifications, the publish-subscribe-collect model can be implemented for the MANET environment.

9. CONCLUSIONS

This paper clearly demonstrates that data sharing coordination can be reasonably implemented on top of message passing. Our initial investigation mapping LIME to of traditional publish-subscribe revealed a path for extending publish-subscribe with a stateful operation, namely *collect*. The resulting model more efficiently supports data sharing applications, is flexible enough to support a wide range of applications, and is stateless itself therefore remaining true to the stateless, message passing model.

10. ACKNOWLEDGMENTS

The authors wish to thank Andrea Bogni for his contributions to the mapping of LIME to traditional publish-subscribe during his master thesis studies at Politecnico di Milano, Italy.

11. REFERENCES

[1] I. M. Arntzen and D. Johansen. A stateful and open publish-subscribe structure for online marketplalces. In *Proc. of the 25th Int. Conf. on Distributed Computing Systems Wkshp.s*, pages 431–437. IEEE Computer Society, 2005.

[2] N. Busi and G. Zavattaro. Publish/subscribe vs. shared dataspace coordination infrastructures. In *Proc. of 10th IEEE Int. Wkshp.s on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*, pages 328–333. IEEE Press, 2001.

[3] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 19(3):332–383, 2001.

[4] K. M. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, 1985.

[5] G. Cugola and G. P. Picco. REDS: a reconfigurable dispatching system. In *Proc. of the 6th Int. Wkshp. on Software engineering and middleware (SEM)*, pages 9–16, New York, NY, USA, 2006. ACM Press.

[6] A. J. Demers, J. Gehrke, M. Hong, M. Riedewald, and W. M. White. Towards expressive publish/subscribe systems. In *Proc. of the 10th Int. Conf. on Extending Database Technology (EDBT)*, pages 627–644, 2006.

[7] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, 2003.

[8] E. Fidler, H.-A. Jacobsen, G. Li, and S. Mankovski. The PADRES distributed publish/subscribe system. In *Int. Conf. on Feature Interactions in Telecommunications and Software Systems*, pages 12–30, 2005.

[9] D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

[10] D. Heimbigner. Adapting publish/subscribe middleware to achieve Gnutella-like functionality. In *Proc. of the 2001 ACM Symp. on Applied computing (SAC)*, pages 176–181, New York, NY, USA, 2001. ACM Press.

[11] G. Mühl, L. Fiege, and P. R. Pietzuch. *Distributed Event-Based Systems*. Springer, Aug. 2006.

[12] A. L. Murphy, G. P. Picco, and G.-C. Roman. LIME: A coordination model and middleware supporting mobility of hosts and agents. *ACM Transactions on Software Engineering and Methodology*, 15(3):279–328, 2006.

[13] G. A. Papadopoulos and F. Arbab. Coordination models and languages. *Advances in Computers*, 46:330–401, 1998.

[14] M. Spezialetti and P. Kearns. Efficient distributed snapshots. In *Proc. of the 6th Int. Conf. on Distributed Computing Systems (ICDCS)*, pages 382–388, 1986.