On Adding Replies to Publish-Subscribe

Gianpaolo Cugola
Dip. Elettronica e Informazione
Politecnico di Milano, Italy
cugola@elet.polimi.it

Matteo Migliavacca
Dip. Elettronica e Informazione
Politecnico di Milano, Italy
migliava@elet.polimi.it

Alessandro Monguzzi
Dip. Elettronica e Informazione
Politecnico di Milano, Italy
alessandro@monguzzi.org

ABSTRACT

Recently, the publish-subscribe communication model has attracted the attention of developers as a viable alternative to traditional communication schemas, like request/reply, for the flexibility it brings to the architecture of distributed applications, by allowing components to be easily added or removed at run-time. At the same time, first experiences in building complex distributed applications using such model point out how it is often hard to live without a request/reply facility.

We started from this consideration to introduce replies into the publish-subscribe model in a way that could minimize the impact on the positive characteristics of the model. In this paper we describe the resulting model and present four protocols to implement it, comparing them through the analysis of the results we gathered in running a large testbed on the PlanetLab network.

Categories and Subject Descriptors

C.2.4 [**Computer Communication Networks**]: Distributed Systems

General Terms

Algorithms, Performance, Measurement

Keywords

Publish-subscribe, Replies, Content-Based Routing

1. INTRODUCTION

Publish-subscribe applications are organized as a set of distributed components, which interact by *publishing* messages and by *subscribing* to the messages they are interested in. A component of the architecture, the *message dispatcher*, usually part of a publish-subscribe middleware infrastructure, is in charge of collecting subscriptions and routing messages from publishers to the interested subscribers.

The communication and coordination model that results from this schema is inherently *asynchronous*, because publishers and subscribers operate in parallel without synchronizing during communication; *multi-point*, because messages are sent to all the interested components; *anonymous*, because publishers do not need to know the identity of subscribers, and vice versa; *implicit*, because the set of message recipients is determined by the subscriptions, rather than being explicitly chosen by the publisher; and *stateless*, because messages do not persist in the system, rather they are sent only to those components that have subscribed before the messages are published. This results in a strong decoupling between publishers and subscribers, which greatly reduces the effort required to modify the application architecture at run-time by adding or removing components. A very positive result from a software engineering point of view, which justifies the popularity of this style of communication.

At the same time, every developer who tried to use a publish-subscribe middleware to build complex distributed applications, has experimented how it is often hard to implement complex interactions among a large set of distributed components by using publish-subscribe alone [2, 3, 5, 11]. Not only it is often useful to explicitly address a single component, but also the asynchronous and unidirectional nature of publish-subscribe reduces its applicability.

If the first limitation can be easily overcome by using one among the many unicast communication facilities available (e.g., RPC, RMI, or Internet sockets), the other is harder to remove. More specifically, in many situations publishers need a reply from the recipients of their messages, while other times it is useful to be acknowledged when messages have been received and processed, e.g., when it is known that new components, which must be part of future interactions, will be instantiated during the processing of those messages.

In this paper we focus on this issue by proposing an extension to the publish-subscribe model to allow subscribers to reply to the messages they receive. Our proposal differs from existing ones, such as JMS [6], because it is better integrated into the publish-subscribe model and easier to use for application developers. It results in a model that joins the typical benefits of the publish-subscribe model with those of the request-reply model. At the same time, the major contribution of this work is not in the proposed extension but in the thorough analysis we provide about the different mechanisms that can be used to introduce replies into a publish-subscribe middleware that adopts a *distributed message dispatcher*.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
DEBS '07, June 20–22, 2007 Toronto, Ontario, Canada
Copyright 2007 ACM 978-1-59593-665-3/07/03 ...$5.00.

Indeed, current research on publish-subscribe has focused on improving scalability by distributing the message dispatcher. A distributed dispatcher is organized as a set of *brokers*, connected in an overlay network, which cooperate to collect subscriptions and route messages. In our work we assume this architecture as a starting point and define four different protocols that can be used to route replies back to publishers. These protocols differ in the performance they can offer and in the guarantees they can provide to the application programmer. We studied both aspects by implementing the four protocols within the publish-subscribe middleware REDS [4] and deploying it on the PlanetLab network [1], building a large distributed testbed.

The remainder of this document describes the outcomes of our study. In particular, Section 2 describes the communication model that results by adding replies to publish-subscribe and how this model can be implemented in a distributed scenario. Section 3 compares four different protocols to route replies back to publisher. Section 4 places our contribution in the context of related work. Finally, Section 5 provides some conclusions and identifies possible future work.

2. REPLIES: MODEL AND IMPLEMENTATION

Adding replies to publish-subscribe involves two steps: defining the communication model we want to realize and implementing it. At the modelling level we focus on how to add replies to the publish-subscribe model of communication without changing its nature. At the implementation level we focus on the algorithms and protocols to route replies back to publishers, keeping in mind that we are interested in a specific scenario, that of a distributed dispatcher. The next section explores the former issue, while Sections 2.2 to 2.4 explore the latter.

2.1 The communication model

If we look at the various models of communication as a spectrum ranging from pure point-to-point message passing as enabled by Internet sockets, to RPC/RMI, up to the most advanced models, like message queuing or those based on shared data spaces, we may observe that the publish-subscribe model occupies an extreme of this spectrum, more or less at the opposite side with respect to RPC/RMI. While the latter is synchronous, point-to-point, enables a bidirectional exchange of information, and requires the caller to explicitly know the identity of the callee, the publish-subscribe model is asynchronous, multi-point, unidirectional, stateless, anonymous, and adopts an implicit addressing schema, where the destination of messages is determined solely by the subscriptions issued before the message was published.

Everyone having good experience in using the publish-subscribe model acknowledges that they are precisely these characteristics that provide the strong decoupling among communicating parties that is fundamental to cope with the dynamism of modern distributed applications. The components of an application adopting a publish-subscribe model of communication may be easily added, removed, or even moved from host to host at run-time with a minimal impact on the other components. Starting from this consideration, our goal is to introduce replies into the publish-subscribe model in a way that would maintain as many of those characteristics as possible.

Our proposal consists of extending the publish-subscribe model by adding a `reply` operation and two special messages: `Repliable` and `Reply`. Every subscriber that receives a `Repliable` message is expected to provide a (single) `Reply` back to the publisher through the `reply` operation. At the publisher side, we introduce two different primitives to collect replies: one to get them one by one as they arrive, the other to collect them all at once. A further primitive, the `hasMoreReplies`, allows to know when new replies are already available at the publisher's side. These operations are summarized in Figure 1.

The resulting model is still multi-point, stateless, anonymous, and uses an implicit addressing schema based on subscriptions. The added primitives support a bidirectional flow of information: from publishers to subscribers and back to publishers. Moreover, depending on the operation used to receive replies back, the new model supports both a synchronous and an asynchronous style of interaction. In particular, the `getReplies` operation allows the publisher to synchronize with the subscribers, waiting until all of them receive the published message, process it, and reply back. Conversely, by combining the `getNextReply` and the `hasMoreReplies` operations, it is possible to implement an asynchronous style of interaction, without the need of suspending the publisher waiting for replies. Even in this asynchronous case, it is possible to know when the last reply has been received, i.e., when the call to the `getNextReply` operation returns `null`.

As a final remark, we observe that reasoning at the model level, we do not considered the issue of reliability. The implementation of the model must put in place appropriate mechanisms to cope with lost replies. We will come back on this issue in the remainder of the paper.

2.2 Managing replies as out-of-band messages: The *OBU* and *OBT* protocols

The simplest approach to implement the model we outlined above is to introduce reply management on top of an existing middleware, sending replies back to the publisher as out-of-band messages, i.e., as UDP packets or opening a TCP connection.

This approach does not require to modify the publish-subscribe middleware and, in principle, can be applied to any existing system[1]. This explains why it is the approach usually adopted by developers who need to implement a bidirectional interaction in a publish-subscribe application.

For our own purposes, we choose REDS as the underlying publish-subscribe middleware. It is an open-source system featuring a modular design: something not particularly relevant at this stage, but important when reply management has to be introduced into the routing kernel, as described in Section 2.4.

REDS provides a distributed dispatcher built as an overlay network of interconnected brokers. Application components access the publish-subscribe services provided by REDS using an object that implements the `DispatchingService` interface. It acts as a proxy to the REDS dispatching network, hiding all the details about routing and distribution. Figure 2 shows the main methods provided by the

[1]Another approach that does not require to modify the publish-subscribe middleware uses standard publications as replies. It will be explored in the next section.

Operation	Description
`void reply(Repliable m, Reply r)`	Replies to the previously received `Repliable` message `m`.
`Reply getNextReply(Repliable m)`	Returns next reply available for `m`. Returns `null` if all replies have been already returned. Suspends the caller if no replies are available, yet.
`boolean hasMoreReplies(Repliable m)`	Returns true if there is one or more replies already available for `m`, false in the other cases.
`Reply[] getReplies(Repliable m)`	Returns the whole set of replies sent back in response to `m`. Suspends the caller until all replies have been collected.

Figure 1: Main operations added to the publish-subscribe model to send and receive replies.

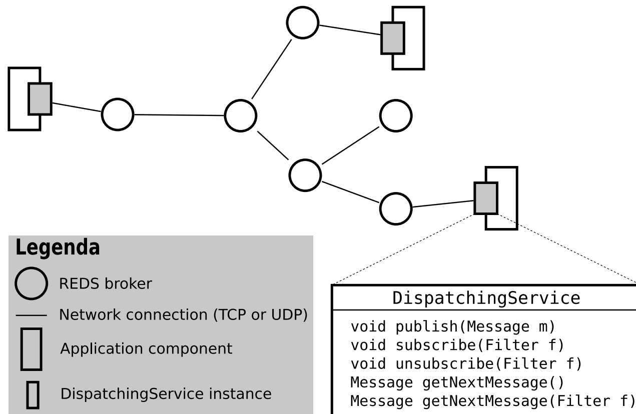


Figure 2: An overview of REDS from an application developer's point of view.

`DispatchingService` interface.

To add replies on top of REDS, we built a `ReplyEnabledDispatchingService` interface, which extends the `DispatchingService` by adding the operations to send and receive replies listed in Figure 1. Being interested in experimenting with out-of-band replies, we provided two different implementations of the `ReplyEnabledDispatchingService` interface, one using UDP packets to route replies back to the publishers, the other using TCP connections. In the following we refer to the resulting protocols as *OBU* (Out-of-Band Udp) and *OBT* (Out-of-Band Tcp), respectively. In both cases the replier needs the IP address of the publisher, which is added to the `Repliable` message.

The most critical point we had to solve in implementing our model with the *OBU* and *OBT* protocols, was how to detect that all replies for a given message have been collected. A fundamental problem to implement the `getNextReply` and `getReplies` methods. In fact, in a publish-subscribe middleware, only the dispatcher knows the number and identity of the components subscribed to each message. The publisher ignores this information. Consequently, we were forced to adopt an indirect mechanism based on a timeout to estimate when all replies have been collected.

Our prototype is parameterized by this timeout τ, representing the time (in milliseconds) to wait for new replies. If the publisher does not receive any reply for τ milliseconds it decides that no more replies will arrive. This approach assumes that each subscriber processes incoming `Repliable` messages, replying immediately. The smallest value for τ can be determined by taking into consideration the expected time for routing messages, processing them, and sending replies back, and it is a compromise between the need of keeping this timeout as short as possible and the need of not loosing replies.

Observe also how the use of a timeout accounts for the case of loosing replies due to failures at the networking or application level. In fact, it avoids the publisher to block forever waiting for replies that were lost during their travel back. We will come back on this issue in Section 3.2.

2.3 Managing replies as standard messages: The *PUB* protocol

An alternative approach to implement our model on top of an existing publish-subscribe middleware uses standard messages as replies, hence the name *PUB* for the resulting protocol. The idea is to take advantage of special subscriptions to guarantee that replies are routed to the right component (i.e., the publisher of the original message the replies refers to) and only to it.

The precise mechanism used to route replies back depends on the type of publish-subscribe middleware in use. In general, each component requiring a reply to the messages it publishes has to issue a special subscription which uniquely identifies itself. This special subscription has to be performed prior to issuing any message that requires a reply e.g., at startup. The nature of such subscription varies depending on the nature of the publish-subscribe middleware, i.e., topic (or subject) based and content based [5, 11].

- In the former case, a component C willing to receive a reply to some of the messages it publishes, creates a globally unique topic t (e.g., by using its own identifier, if available) and subscribes to it. When a distributed dispatcher is used, this step has to complete before the first `Repliable` message can be published. That is, the publish-subscribe middleware must configure its routing tables to correctly route messages addressed to the new topic before it can process the replies. When C needs to publish a `Repliable` message m, it adds t's identifier to the message and publishes it. A component C_1 receiving m publishes its reply as a message addressed to t, being guaranteed that it will be routed by the dispatcher to the right component, i.e., C.

- In the latter case, C generates a globally unique identifier id and subscribes to the messages that include such identifier in their body. When C needs to publish a `Repliable` message m, it adds the identifier id to m (e.g., as a special field). Components receiving m extract the identifier id, add it to their replies, and publish them. It is dispatcher's responsibility to address such replies to the only subscriber available, i.e., C.

Based on these considerations, we implemented a reply management layer on top of REDS in a way similar to what done before. We provided a new implementation for the `ReplyEnabledDispatchingService` interface, using the second approach above, the one tailored to content-based systems like REDS, to implement the correct routing of replies. In particular, we used the identifier of the publisher, which was already provided by REDS, as the special content to be added to replies in order for REDS to correctly route them back to the publisher.

As in the previous case, we had to overcome a limitation that is typical of any solution built "on top" of an existing publish-subscribe middleware: the fact that the number of recipients of a message (and consequently the number of expected replies) is hidden to the publisher. We adopted the same approach described before, using a timeout τ.

If we compare the resulting PUB protocol with those using out-of-band messages, we notice that PUB requires each component to subscribe to a different, unique, filter, if it wants to receive replies to the messages it publishes. This increases the total number of subscriptions up to a level that could be hard to manage for the underlying publish-subscribe middleware, especially in an Internet-wide scenario, with millions of potential components requiring replies to the messages they publish. We will provide more details on this issue in Section 3.

2.4 Managing replies into the routing kernel: The KER protocol

The three protocols we described so far are characterized by the impossibility of precisely determining when all replies have been collected. To overcome this limitation it is necessary to add the reply management layer into the routing kernel of the publish-subscribe middleware in use. This way, the module in charge of managing replies can cooperate with the module in charge of routing messages to determine the exact number of recipients of each message and consequently the number of expected replies. Clearly, this require full access to the internals of the publish-subscribe middleware in use.

In our case, this was not a problem: not only REDS is an open source project, but it also adopts a modular architecture that simplifies the job of adding new functionalities into the routing kernel of each broker. Thus we added a new module to each REDS broker, which implements a *convergecast* [10] routing schema for replies. It tracks `Repliable` messages while they flow along the REDS overlay network, from publishers to subscribers, and uses this soft state to route replies back and to determine the exact number of replies to wait for. It uses the unique identifier of each `Repliable` message, which is copied by the `reply` operation into replies, to correlate them with the message they refer to.

More specifically (see Figure 3), the new `ReplyManager` module we added to REDS is in charge of routing replies back to the publisher. When a `Repliable` message msg reaches a broker B, it is processed by the `Overlay` component (which hides all the details about distribution) and sent to the `Router`. The `Router` invokes the `RoutingStrategy`, which forwards the message along the REDS overlay toward the subscribers and returns an integer n: the number of neighbors the message was forwarded to. This key information, together with the identifier id_{msg} of msg and the identifier id_{src} of the neighbor msg comes from (i.e., the publisher at the first hop or another broker at further hops) is passed to the `ReplyManager`, which stores it into a `ReplyTable`.

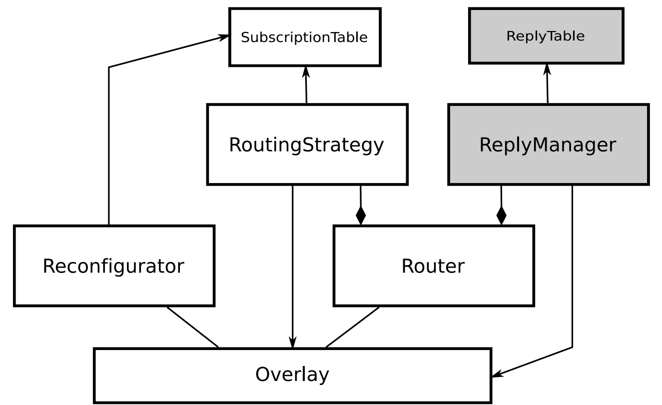


Figure 3: A high level view of the internal architecture of a REDS broker: In grey the new components added to manage replies.

When a subscriber replies to msg, the id_{msg} is copied, by the `reply` operation, into the `Reply` object, which is then sent to the broker the replier is attached to. When this `Reply` rpl reaches the broker B, it flows from the `Overlay` to the `Router`, which this time invokes the `ReplyManager` directly. The `ReplyManager` uses the message identifier id_{msg} saved into rpl to determine, from the `ReplyTable`, the neighbor that has to receive the reply. Moreover, it checks if rpl was tagged as *last*. This is a flag used by each broker to inform the next one along the route toward the publisher about the fact that all replies coming from downstream components have been collected. Accordingly, when the `ReplyManager` receives a reply rpl tagged as *last* it decreases the number of expected replies stored into the `ReplyTable` for that particular message. If the resulting value is greater than zero it removes the flag before forwarding rpl, otherwise it leaves it there and deletes the corresponding record from the `ReplyTable`.

At the publisher site and particularly into the `getNextReply` and `getReplies` operations of the `ReplyEnabledDispatchingService`, the presence of the *last* flag determines when the last reply for each message arrives. Without requiring approximate solutions based on timeouts, like those used in the previous protocols.

The case of replies that got lost due to faults at the networking or application layers is solved by taking advantage of the mechanisms offered by REDS brokers to detect when a neighbor has failed. When this happens the `ReplyManager` is informed and acts as if the last reply from that neighbor had arrived, reporting the fault upstream through an appropriate empty reply flagged as *last* and *failed*. Both flags continue to propagate upstream up to the publisher where the `ReplyEnabledDispatchingService` checks if the last reply is also flagged as *failed* and in this case returns an exception to the publisher waiting for replies.

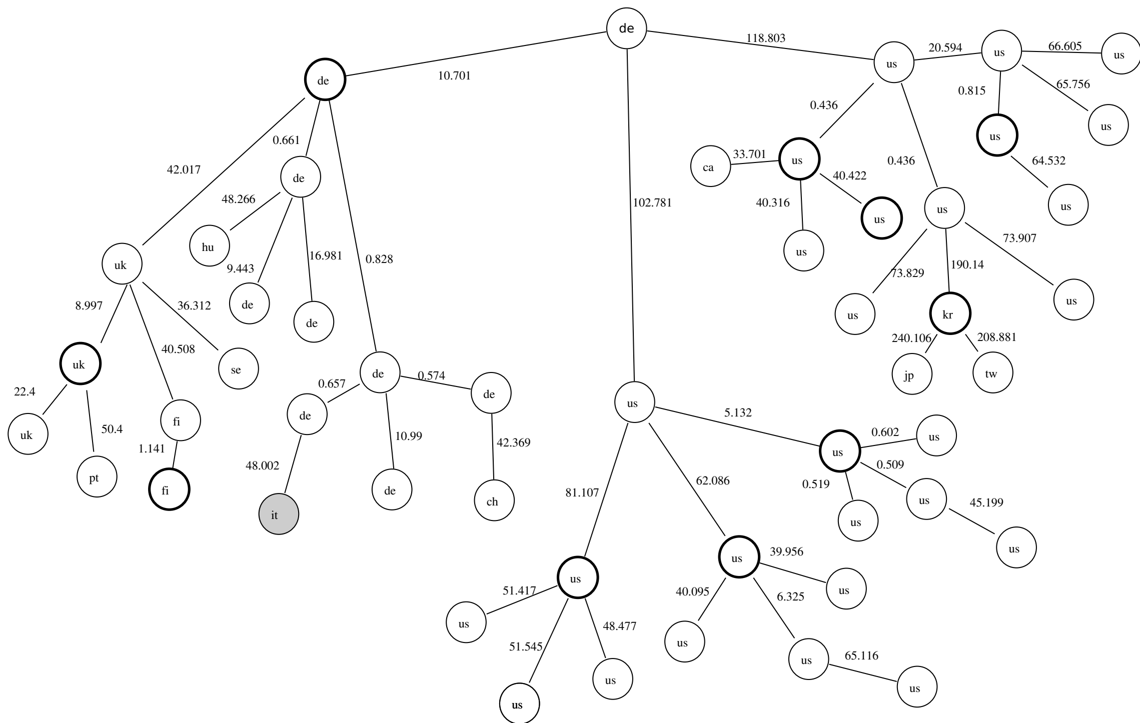


Figure 4: The dispatching network used for the tests.

3. EVALUATION

In this section we evaluate and compare the four protocols described above both looking at the raw performance they offer in Section 3.1, and examining other qualitative properties in Section 3.2.

3.1 Performance analysis

To compare the four protocols described above from a performance standpoint we took advantage of the PlanetLab [1] network, which allows to replicate the same situations that can be found in real, large scale, publish-subscribe scenarios.

In particular, we selected 50 nodes from PlanetLab that we found being stable enough to complete our tests, and installed a REDS broker on each node, connecting them in an overlay network, which was never changed for all the experiments. In defining such overlay we tried to match the topology of the physical network as much as possible, by directly connecting those nodes that exhibited the lowest latencies. The resulting topology is shown in Figure 4. For each node it shows the country where the node is located together with the average latencies with its neighbors, measured by S^3 [14] in the same week of our final tests.

To test the four protocols, we attached a single "publisher" to the broker at the root of the dispatching network, which sends messages and waits for replies, measuring the time elapsed from the publishing to the arrival of each reply. Each node also runs ten "repliers" attached to the broker running on the same node. Each replier may use any of the four protocols described above to reply and it is subscribed to different messages to allow for six different interaction scenarios: a single or ten repliers attached to the broker shown in gray in Figure 4, a single or ten repliers for each of the ten brokers highlighted using a thick line in Figure 4, a single or ten repliers for each of the 50 brokers. In the following we refer to these scenarios as "1x1", "10x1", "1x10", "10x10", "1x50", and "10x50", respectively.

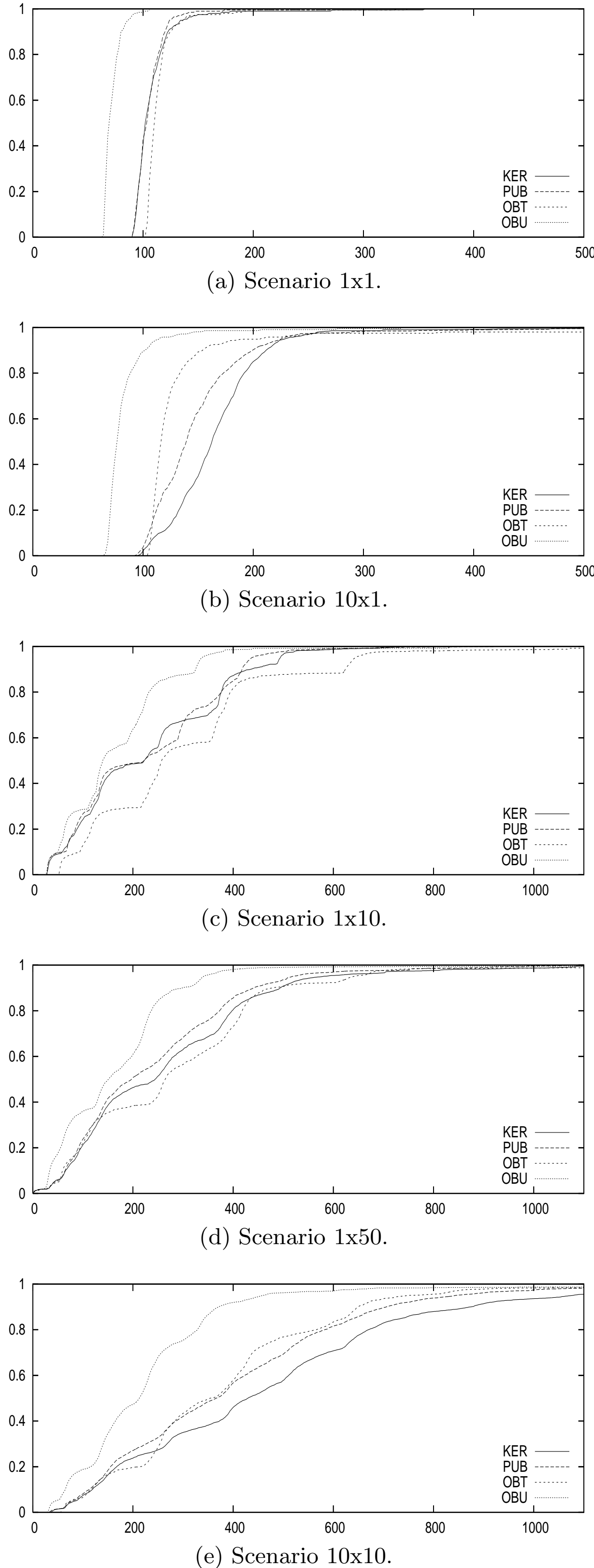
To test the different protocols under traffic, each node also runs an additional client, the "loader", attached to the broker running on the same node, which generates random traffic towards the other nodes. The loads we tested were 0, 0.5, and 1 message per second per loader (i.e., per node), with each message going, on average, to 5% of the other loaders.

At each run we choose the interaction scenario and the load to test and let the publisher send 200 messages for each of the four protocols OBT, OBU, PUB, and KER. To ensure that a transient slow down of the PlanetLab network, or of the Internet as a whole, impact equally on all the protocols, we interleaved them by first sending a message to be replied through the OBU protocol and waiting for all the replies to come, then sending the same message but this time to be replied through the OBT protocol, and so on for the four protocols, repeating this cycle 200 times.

As a final remark, we warn the reader that different runs testing different interaction scenarios or loads, should not be compared directly. Indeed, each run lasts from minutes to hours: the conditions of the PlanetLab network could have changed greatly in such interval.

3.1.1 *Non-congested scenarios*

The first characteristic we evaluate is the round trip time (rtt) required to publish a message and receive the corresponding replies. To exclude possible saturation effects we present the data measured in scenarios free from any additional traffic (i.e., with load=0) and with a maximum num-



(a) Scenario 1x1.

(b) Scenario 10x1.

(c) Scenario 1x10.

(d) Scenario 1x50.

(e) Scenario 10x10.

Figure 5: Cumulative distribution of rtt (percentage of replies vs rtt in ms), load=0.

ber of 100 repliers. We will examine the most congested scenarios in Section 3.1.2.

To have a precise view of the behavior of the different protocols, each graph depicts the cumulative distribution of the rtt over the entire set of 200 messages sent by the publisher for each run. Accordingly, the x-axis reports the values of rtt, while the y-axis reports the percentage of replies received. Each point of a curve represents the percentage of replies received (read on the y-axis) whose rtt was less than or equal the value read on the x-axis.

Figure 5(a) shows the rtt measured for the case of a single replier. The *OBU* protocol is the fastest one, followed by *PUB* and *KER*, which almost coincide, and then by *OBT*, which is the slowest protocol.

Besides these raw differences, the figure also shows many interesting facts that help us understanding the general behavior of the four protocols. First of all, the values measured for the *PUB* and *KER* protocols are almost identical in this scenario and also very similar in all the other scenarios that we examined. Three facts contribute to this: first, replies in both approaches experience exactly the same network latencies as they are routed through the same overlay links; second, both protocols require brokers along the route to process the replies, incurring in a further delay at the application layer; third, the processing at each broker is similar, with the *KER* protocol looking at the reply table while the *PUB* protocol uses the subscription table.

With respect to the last observation, the informed reader could observe that in principle the reply table should be much smaller than the subscription table. Moreover, while the former allows a direct access (indexed through the message identifier), the latter requires a slower, content-based matching. This should allow the *KER* protocol to outperform the *PUB* one. While this is true in theory, in practice we run our tests in a conservative situation, with a few subscriptions, which results in the two protocols perform similarly. As a further note related with this issue, in Section 2.3 we mentioned that the *PUB* protocol requires additional special subscriptions to guarantee correct routing of replies toward the publishers, which is very likely to have a negative impact on performance. Again, this is true in theory but in our scenarios we have a single publisher and consequently a single additional subscription. Notice that we could have introduced more publishers but the need of interleaving the tests of the four protocols on each run would not have allowed us to test the impact of such additional subscriptions on the *PUB* protocol taken in isolation.

By looking at the graphs we may also observe that the fastest *PUB* reply arrived 92ms after the publication, after 103ms we received half of the replies, and almost all of them (97%) after 150ms. By halving the second of these numbers we may estimate the average time each message takes to go from the publisher to the replier: approximately 50ms. This time is the same for each protocol. Accordingly, the *OBU* protocol, whose first reply arrived after 64ms, while half of the replies are delivered before 70ms, reaching 97% at 90ms, takes approximately 20ms to route replies back to the publisher. This is a much shorter time than that measured for the *PUB* and *KER* protocols, which is reasonable if we consider that the *OBU* protocol experiences a shorter network latency as its replies does not follow the overlay, and, most importantly, they are not processed by the intermediate brokers. Finally, the estimate of 20ms for the *OBU*

replies to come back is coherent with the times measured for the OBT protocol, whose first reply arrived after 104ms, while half of them arrived after 111ms, reaching 97% of the replies after 140ms. In fact, on average OBT should require 50ms to reach the replier, plus 3 times the network latency, i.e., $20*3 = 60ms$, to open the TCP connection and deliver, in a single packet, the reply, which totals 110ms; a number very close to that measured.

Figure 5(b) shows the rtt distribution in the scenario with 10 repliers attached to the same gray node of Figure 4. Here we start seeing the first queuing effects: for OBU and OBT we observe a slight increase in the median and a relevant increase on the rtt for the late replies due to queuing. The same effect, but in a more severe form, affects the PUB and KER protocols. This can be explained by observing that in those cases queuing occurs not only at the network and operating system layers, but also at the application layer, within the brokers that have to process the replies to route them back. This effect is exacerbated by the particular condition of the gray node where the replier is located, which is relatively close to the publisher in terms of network latency (approximately 20ms, as we saw before), but far in terms of number of hops on the overlay: five. This explains why in this scenario, where the throughput of REDS brokers becomes a bottleneck, the OBT protocol performs slightly better than PUB and KER.

Figure 5(c) depicts what happens when a single replier is placed on each of the ten, thick nodes of Figure 4. The graph shows some interesting insights. In particular, the OBU line is not as steep as in the case of a single replier, showing small plateaux corresponding to intervals of time where (almost) no messages arrived. Noticing that these plateaux are placed around multiples of 10% of delivery (that corresponds to one reply, since in this scenario each message fires 10 replies) and correlating them with the latencies measured on PlanetLab, it is possible to conclude that each step following a plateaux corresponds to the instant in which the reply from a certain node arrives. The same behavior affects the OBT protocol but with higher rtts due to the handshake required to open the TCP connection. Moreover, as time progresses the OBT performance get worse with respect to OBU. This can be explained by observing that nodes that are reached later on the overlay, if this matches the physical network like in our case, are also further away (in network latency term) from the publisher, thus the opening of the TCP connection takes longer.

If we look at the PUB and KER protocols we may observe that in this case they perform better than OBT. This can be explained by looking at the test topology. The ten thick nodes where repliers are located are, on average, at less hops on the overlay than the single gray node used in the previous scenario, thus incurring in less processing time at the application layer. They are instead further away from the publisher latency-wise, thus making the handshake of the underlying TCP transport more expensive. As in the previous scenarios, PUB and KER are very close together.

Figure 5(d) graphs the rtts measured in the scenario with a single replier on each node, showing a behavior similar to that exhibited in the 1x10 scenario; while the behavior of the four protocols in the 10x10 scenario, shown in Figure 5(e), is a combination of the trends of the 1x10 and 10x1 scenarios.

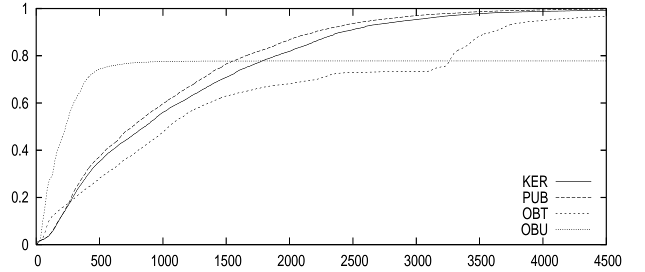


Figure 6: Cumulative distribution of rtt (percentage of replies vs rtt in ms) in the 10x50 scenario, load=0.

3.1.2 *Congested scenarios*

All the scenarios presented so far were not affected by problems of congestion, as the total number of replies to manage was limited and they were spread temporally in small bursts occurring when the test message reaches each replier. The situation changes dramatically in the 10x50 scenario, where at each hop traveled by the test message along the dispatching overlay 10 replies for each of the reached nodes are fired back, overloading the publisher.

The impact of such behavior on the distribution of the rtt measured for the different protocols is shown in Figure 6. As expected, the trends it shows are quite different with respect to those measured in the previous scenarios.

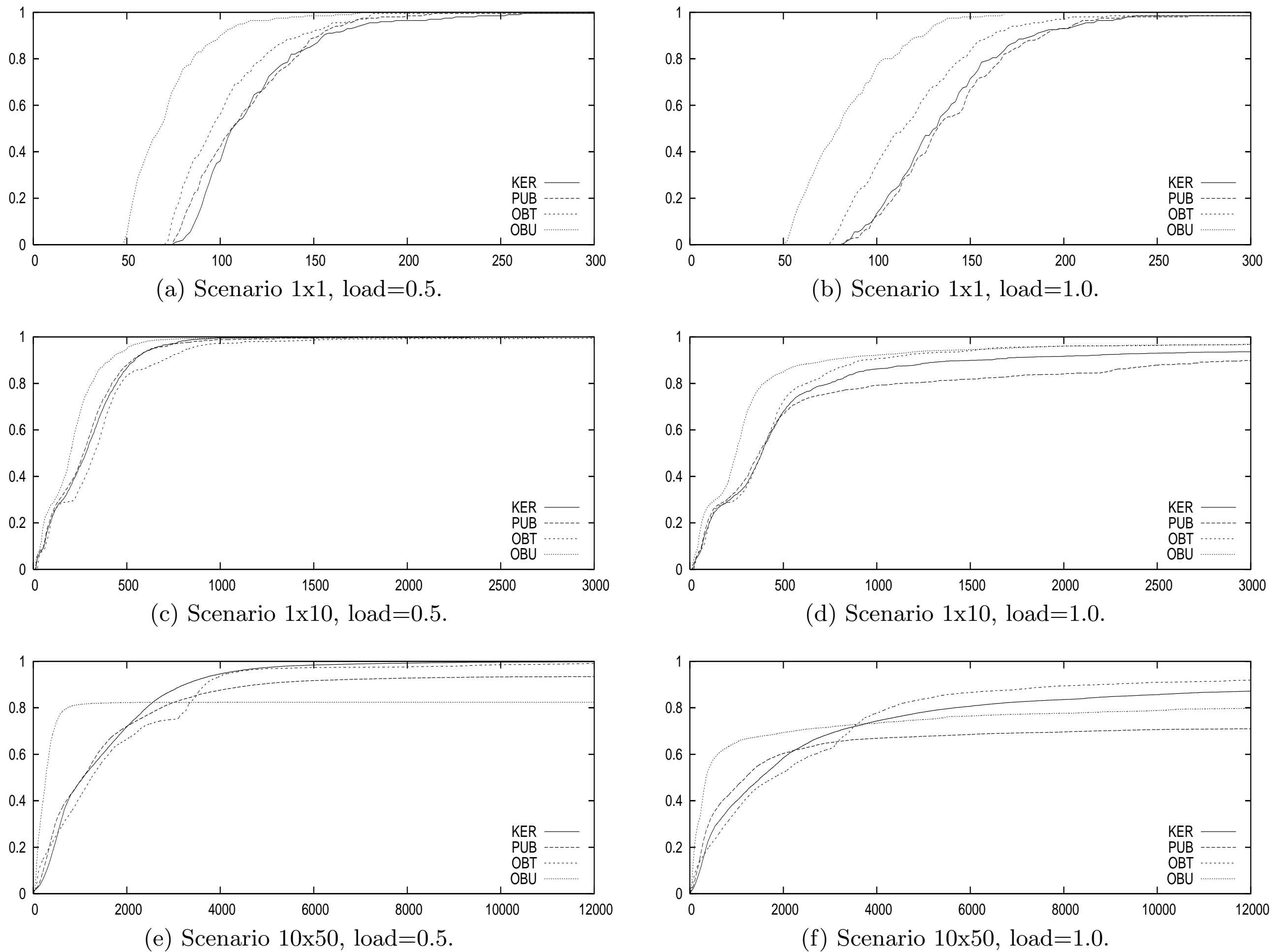
The OBU protocol confirms its lead converging in about 500ms. Unfortunately, it also shows its unreliability: the storm of replies toward the publisher cause massive dropping of packets at the networking layer, with a final delivery limited to 78% of the total number of expected replies. Notice that this behavior, while not evident by looking at the graphs, was also present in the previous scenarios. The OBU protocol missed 226 replies in the 10x10 scenario, 46 in the 1x50, 9 in the 1x10, and one in the 1x1 scenario.

The behavior of the OBT protocol is also quite characteristic. At the beginning the delivery grows straight, meaning that the number of replies delivered at each time interval is constant up to reaching 73% of delivery, only slightly below the OBU result. At this point very few replies are received for about 1.5 seconds when (at 3s from the beginning) the TCP timeout elapses and retransmission of lost packets begins. During this retransmission phase other collisions occur and the delivery does not reach 100%, reaching 97% after 4.5s from the beginning and stopping there up to 9s (as TCP doubles the timeout for each unsuccessful retry) when a second retransmission phase starts. This sequence of retransmission-collision-timeout is repeated many times: the last OBT reply arrived almost one minute after the initial publication.

The PUB and KER protocols exhibit the best performance in this scenario. The figure shows a smooth growth of the delivery with almost 100% of replies delivered in 4.5s for both.

In the scenarios we considered so far, the only messages flowing through the dispatching network were the test messages sent by the publisher and the corresponding replies. In Figure 7 we analyze how additional traffic flowing on the REDS overlay affects the performance of the four protocols we consider.

In particular, Figures 7(a) and 7(b) describe what hap-



(a) Scenario 1x1, load=0.5. (b) Scenario 1x1, load=1.0.

(c) Scenario 1x10, load=0.5. (d) Scenario 1x10, load=1.0.

(e) Scenario 10x50, load=0.5. (f) Scenario 10x50, load=1.0.

Figure 7: Cumulative distribution of rtt (percentage of replies vs rtt in ms) in loaded scenarios.

pens in the 1x1 scenario with a load set to 0.5 and 1 messages per seconds (per loader), respectively. Here we see that the PUB and KER protocols are most influenced by this additional traffic, losing their second position in favor of OBT. This is natural as both protocols require replies to be processed by brokers along the route toward the publisher. The more these brokers are loaded the worse they perform. We may also notice how the KER protocol gains a little with respect to PUB. This trends, which is confirmed by the measurements we performed in other scenarios, is a result of the fact that REDS brokers use a different queue for messages and replies, thus inherently privileging the KER protocol, which explicitly uses replies, over the PUB protocol, which uses standard messages as replies.

While the 10x1 scenario is quite similar to the 1x1 one, and it is not shown here, the 1x10 scenario at load 0.5 and 1, shown in Figures 7(c) and 7(d), respectively, shows some peculiarities. In particular, we observe that the small plateaux present in the unloaded chart disappear. This can be explained by observing that under load there is a greater variance in the time required by messages to reach the repliers and consequently it is no more possible to clearly isolate the replies coming from each node. By comparing the two figures we may also observe how the load affects the PUB and KER protocols, which are still ahead of OBT when load=0.5, while loose the lead at load=1. As in the previous scenarios, the KER protocol is less affected than PUB by the traffic.

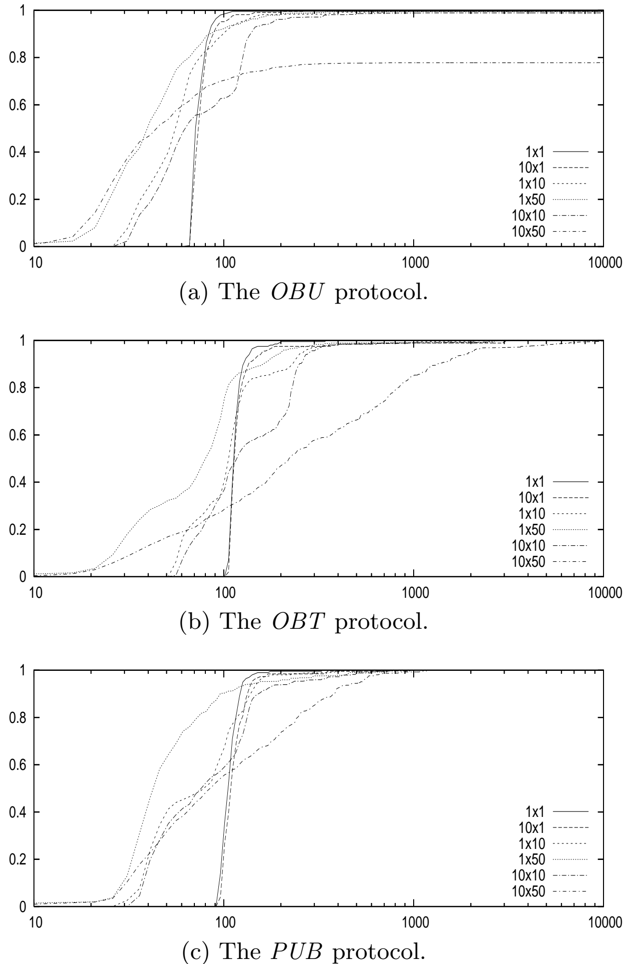
The last scenario we consider involves 10 repliers for each node. With a load of 0.5 messages per second the KER protocol shows its benefits, taking the lead and delivering most replies in less than 3s. At 1 message per second the things changes. By looking at the PUB graph, we may observe how in this situation the REDS middleware reaches its limits. The few CPU cycles guaranteed to each slice on the PlanetLab network do not allow REDS brokers to provide enough throughput to cope with such traffic. This considerably affects the performance of the PUB protocol, but also negatively affects the KER protocol, which is know surpassed by OBT.

3.1.3 The impact of τ on delivery

All the graphs presented so far, were captured by adopting a timeout τ of 30s: a value unbearable for most application settings. Consequently, our next goal was to analyze how the delivery is influenced by the choice of τ, a fundamental

step to decide the minimum value of τ that provides a given delivery.

Figure 8(a) plots the delivery of *OBU* (y-axis) varying τ (x-axis) in the different scenarios we consider. It shows the expected behavior for an unreliable protocol, either it delivers replies quickly or it drops them. From a certain point of view this is a nice behavior. It allows to easily determine a value of τ working in most situations, which in our case equals 200ms. This is fundamental, as in publish-subscribe the publisher usually has no idea about the number of subscribers that will receive the messages it publishes. The value chosen for τ must be able to accommodate most scenarios.



(a) The *OBU* protocol.

(b) The *OBT* protocol.

(c) The *PUB* protocol.

Figure 8: Delivery versus τ (in ms), load=0.

Figure 8(b) shows the behavior of the *OBT* protocol. Here we see a different situation. Indeed, it is quite difficult to find a value for τ that works nicely in all the different scenarios we consider. For the non-congested scenarios a value of 200ms is fine, but this value does not account the 10x10 scenario, which requires at least 400ms to reach a good delivery. Even worse is the case of the most congested scenario, the 10x50, which needs a value of τ in the order of seconds to provide a good delivery.

The last protocol we consider is *PUB* (*KER* does not use timeouts to decide when the last reply has been collected), which combines the positive aspects of the two protocols above, minimizing their weaknesses. Its behavior, shown in Figure 8(c), allows to determine a reasonable value of τ (i.e., approximately 600ms) working in all the scenarios we consider, while providing good delivery ratios.

The considerations developed so far for the unloaded scenarios also apply under load, with *OBT* and *PUB* requiring progressively a larger τ to provide the same performance. For the sake of brevity, we do not report these graphs here.

3.2 Qualitative analysis

In the previous section we compared the four protocols with respect to their performance, here we analyze them with respect to other, non-quantitative qualities they show.

First consideration we can do is related with the guarantees offered by the different protocols. In particular, *KER* is the only protocol that is capable of precisely notifying the publisher when all replies have been collected. All the other protocols have to rely on a timeout to decide when to stop waiting for new replies. This, not only reduces the responsiveness of the system, obliging publishers that want to synchronize with the recipients of their messages (i.e., using the `getReplies` primitive) to wait the entire timeout, but also does not allow to determine when faults occur. The *KER* protocol is the only one being able to determine when the process of collecting replies has ended and to determine if this process has encoutered a fault that resulted in loosing one or more replies.

Another difference among the four protocols has to do with the mutual ordering between replies and subscriptions. Some publish-subscribe middleware, in fact, guarantee that if a component C subscribes to a filter f at time t, it will receive any message matching f that was causally related with messages C sent after t. This temporally correlates the action of subscribing with the action of publishing. Now consider a situation in which a component C_2 receives a `Repliable` message from C_1 and reacts by first subscribing to a filter f then replying. If the reply is routed using the *PUB* protocol implemented on top of a system that satisfy the property above, then C_1 is guaranteed that after receiving the reply from C_2 it can publish a message matching f sure that C_2 will receive it. The same behavior can be offered by a proper implementation of the *KER* protocol. Unfortunately, none of the two out-of-band protocols, *OBU* and *OBT*, can provide the same guarantee without introducing complex ordering mechanisms.

The consideration above can be generalized by observing that the two protocols *PUB* and *KER* usually inherit all the properties offered by the publish-subscribe middleware they integrate into. In particular, some publish-subscribe middleware offer advanced mechanisms to deal with mobile components or to operate in very complex and dynamic scenarios like mobile ad-hoc networks (MANETs) [12] or wireless sensor networks [13]. In similar scenarios it is much easier to implement the *PUB* and *KER* protocols, leveraging the routing mechanisms already offered by a publish-subscribe middleware developed expressly for such scenarios, than to implement the *OBU* and *OBT* protocols with the need of explicitly managing the complexity characterizing point-to-point routing on such scenarios.

As a final remark, we may observe that the *KER* pro-

tocol is the only one that could be profitably extended to include mechanisms to aggregate replies while they route back toward the publisher. This aggregating function could be hard-wired, e.g., by collecting replies in blocks to be compressed and routed back as a single message, or it could be provided by the publisher. As an example of the latter case, we could have a component publishing a `Repliable` message to notify its interest in receiving information about the current temperature. In this case the publisher could provide a function to aggregate replies, sending back only the average temperature measured.

3.3 Concluding remarks

From the quantitative and qualitative evaluations provided in the previous sections we may draw some conclusions about the different protocols we examined.

First of all we may observe that OBU is the fastest protocol among the four we considered, while providing very few guarantees. At the same time, its simple behavior, which either delivers replies in a short time or drops them, simplifies the choice of τ. Waiting more than a certain (short) interval does not pay off. If replies have not been delivered after a short interval they will never be.

The OBT protocol has opposite characteristics: it is reliable but the mechanism of retransmission, which is part of TCP, makes it hard to estimate a good value for τ.

The two in-bound approaches perform equally nicely. In our testbed they are limited by the overhead introduced by the specific publish-subscribe middleware used and by the fact that the PlanetLab nodes are usually very loaded, leaving few CPU cycles to each user. In more controlled scenarios, with nodes entirely dedicated to running the REDS brokers, they would perform much better. At the same time, even if it was not possible to explicitly measure this behavior in our testbed, the performance of the PUB protocol are expected to degrade when the size of the network grows, with larger subscription tables and more components requiring replies to their messages.

With respect to the estimation of τ, the PUB protocol shows a nice behavior, with a slow variance in the time required to deliver replies under very different loads (see Figure 8(c)). Clearly, KER is the only protocol that is not affected by the problem of finding a good estimate to τ, also offering the possibility of processing replies as they are collected.

Finally, in complex scenarios characterized by a strong dynamism at the networking layer, as in MANETs, the PUB and KER protocols may benefit of the mechanisms provided by a publish-subscribe middleware appositely developed for such scenarios to easily route replies back to the publisher.

4. RELATED WORK

The usefulness of replies even in the context of a publish-subscribe model of interaction has been originally noticed in [3]. Since then many distributed applications (if not most) have coupled a publish-subscribe communication facility with a point-to-point one (mainly some form of RPC) but, to the best of our knowledge, very few research papers have reported about the tentative of integrating reply management into publish-subscribe.

In [7] authors describe how to develop a query-advertise system functionally equivalent to Gnutella, the popular peer to peer search engine, starting from a publish-subscribe middleware. Query-advertise systems are a sort of dual with respect to publish-subscribe systems, in which subscriptions carry advertisement of resources, while publications allow to query for matching resources. These systems obviously need some way of returning back the results of queries. In [7], authors propose an algorithm very similar to PUB, while in [8], they cite the possibility of using either PUB or OBT. In both cases no performance numbers are given as the focus of the papers is on the modifications to the matching system required to provide the query-advertise style of interaction.

JMS [6] was the first, largely available publish-subscribe infrastructure, to provide a concept of reply. On the other hand, it does not provide a fully integrated reply mechanism as we propose here, while it merely defines the basic mechanisms to send replies back to the publisher. In particular, JMS messages include two standard fields in their header: the `JMSCorrelationID` can be used in replies to specify the identifier of the original message, while the `JMSReplyTo` field can be used to specify the `Destination` where replies are supposed to be addressed to. A common pattern to manage replies using such fields involves creating a (temporary) destination for replies and adding its reference into the `JMSReplyTo` field of the published message. Depending on weather the destination is a queue or a topic the resulting machinery ends up being similar to OBT or PUB, respectively.

A more integrated approach to extend publish-subscribe with replies is presented in [9]. There the authors provide a fully asynchronous `reply` primitive. At publishing time, the publisher of a message m obtains a handler to a *reply view*. Through such handler it can query the publish-subscribe system to get the replies to m collected up to that time. It may also obtain an aggregate of all the replies collected that far. Subscribers receiving m may reply one or more times to m, or they may completely ignore it. The publisher is also allowed to destroy the reply view associated with m when it is no longer interested in collecting replies. The result is a model very different to that we presented here. The fact that the same subscriber may reply more than once and that a reply view remains active until the publisher explicitly removes it, make this model more similar to long running queries than to a reply management service as we had in mind. Moreover, this way of modelling replies does not try to close the gap between the asynchronous publish-subscribe model and a more traditional, synchronous interaction. Indeed, the fact that nodes are free to choose if replying or not makes the middleware unable to provide any indication about the fact that all replies have been collected. Finally, this model requires the publish-subscribe middleware to keep the state used to route replies back for a unbounded time, something that our approach does not require.

5. CONCLUSION

In this paper we presented an analysis of the alternatives to extend the publish-subscribe model with a reply mechanism. This need was motivated by our own experience in building applications with publish-subscribe and it was already expressed by the publish-subscribe user community as demonstrated by the appearance of patchy, ad-hoc solutions, often without any performance evaluation.

The contribution put forth in this paper includes a precise modelling of a reply-enabled publish-subscribe style, the description of four possible protocols to implement this model

in a fully decentralized way, completed by a throughly characterization of their performance obtained through extensive measurements on PlanetLab.

This integrates nicely with our long term goal of increasing the applicability of the publish-subscribe model in building complex distributed applications by augmenting it with new primitives and mechanisms, as soon as they do not interfer with the positive features it provides.

Acknowledgements

This work was partially supported by the italian National Research Council (CNR) under the IS-MANET project, and by the European Community under the IST-034963 WASP project.

6. REFERENCES

[1] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman. Planetlab: an overlay testbed for broad-coverage services. *SIGCOMM Computer Communication Review*, 33(3):3–12, 2003.

[2] G. Cugola. Tolerating deviations in process support systems via flexible enactment of process models. *IEEE Transactions on Software Engineering*, 24(11), Nov 1998.

[3] G. Cugola, E. D. Nitto, and A. Fuggetta. The jedi event-based infrastructure and its application to the development of the opss wfms. *IEEE Transaction on Software Engineering*, 27(9):827–850, September 2001.

[4] G. Cugola and G. Picco. REDS: A Reconfigurable Dispatching System. In *Proc. of the 6th Int. Workshop on Software Engineering and Middleware (SEM06)*, pages 9—16, Portland, Oregon, USA, nov 2006. ACM Press. Available at `www.elet.polimi.it/upload/cugola`.

[5] P. Eugster, P. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 2(35), June 2003.

[6] M. Hapner, R. Burridge, R. Sharma, J. Fialli, and K. Stout. Java message service specification, April 2002.

[7] D. Heimbigner. Adapting publish/subscribe middleware to achieve gnutella-like functionality. In *Selected Areas in Cryptography*, 2001.

[8] J. C. Hill. An efficient implementation of query/advertise. Technical report, University of Colorado at Boulder, Boulder, Colorado, USA, 2003.

[9] J. C. Hill, J. C. Knight, A. M. Crickenberger, and R. Honhard. Publish and subscribe with reply. Technical report, University of Virginia, Charlottesville, VA, USA, 2002.

[10] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, San Mateo, CA, 1996.

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

[12] C. E. Perkins. *Ad hoc networking*. Addison Wesley, 2001.

[13] C. Raghavendra, K. Sivalingam, and T. Znati, editors. *Wireless Sensor Networks*. Springer, 2006.

[14] P. Yalagandula, P. Sharma, S. Banerjee, S.-J.Lee, and S. Basu. S^3: A scalable sensing service for monitoring large networked systems. In *Proc. of the Workshop on Internet Network Measurement, Pisa*, 2006.