XEP-0040: Jabber Robust Publish-Subscribe

Tim Carpenter
mailto:tim.carpenter@in8limited.co.uk
xmpp:

2004-07-26
Version 0.2

Status	Type	Short Name
Retracted	Standards Track	None

Note: This proposal has been superseded by XEP-0060; please refer to that document for the successor protocol.

Legal

Copyright

Permissions

Warranty

Liability

Conformance

Contents

1 Introduction

Note: This XEP has been superseded by Publish-Subscribe (XEP-0060) [1]; please refer to that document for the successor protocol.
This document introduces and lays out a preliminary protocol for a robust form of publish-subscribe over the Jabber messaging environment -- Jabber Robust Publish Subscribe (JRPS). Implementation issues in the environment are appended, covering Permissioning and Contributions. Both are likely to require separate XEPs, but need to be constructed sympathetically. In creating this addition, I have an underlying philosophy to sustain a "fractal" world of publish-subscribe components, such that a subscriber to a pubsub component may well be a pubsub component in itself, representing its own community of subscribers. This will allow Jabber to support organic scalability found on other platforms.

1.1 Background

Publish-Subscribe and other messaging environments that exist are often classified as providing one or more of the following three levels of service.

1. Best Try, where data may on rare occasions get lost. Small footprints and ultimate performance are the aim where the impact of occasional data loss in business, legal, confidential or other terms is not significant compared with the core priority of performance.
2. Robust, where non-delivery of data can be detected and recovered by recipients and that the sequence, integrity and completeness of data can be ascertained with a high level of confidence. Non-delivery of data would have a significant and lasting negative impact on the quality and integrity of the data.
3. Transactional, where there is an absolute, mission-critical need to ensure that all communication flow is guaranteed and if problems occur during a set of connected steps, then the situation can be rolled back (reversed) to the state before the operation commenced.

This document concerns itself with level 2 Publish Subscribe -- "Robust".

1.2 Positioning

JRPS is required in environments where there is a higher demand for guaranteed delivery in high throughput, low latency environments where data has value and can contain business intelligence, but does not demand a full transactional (e.g. 2-phase commit) strength environment.
Such environments often exercise business logic upon data received, so the notion of updates

[1] XEP-0060: Publish-Subscribe <https://xmpp.org/extensions/xep-0060.html>.

to all or part of data, the expression of the definitive, full compliment of a particular set of related data, the correction of data in full or in part and the notification that data is no longer valid needs to be supported. The existing type=”set”, though very suitable in a wide range of applications, does not provide suitable granularity in all environments.
Robust environments require that a receiver can tell when data has been lost and that a receiver also has the means to request the repair of any gaps efficiently. This must be done whilst keeping delay or disruption to ongoing data flow to a minimum. Jabber does not provide the means to detect or repair gaps, and traditional ACKing of each packet is slow and costly.
It would be advantageous to permit forms of permissioning and access control upon data that has value. Such permissioning and control should not be overly burdensome on the rapid transmission of data. It should allow a suitable level of abstraction to keep changes to a data item”s expression of permission coding/level to a minimum, to avoid the need for excessive changes to such codes. Abstraction will also permit permission coding to be kept compact, as it will, in effect, be tokenised.
JRPS then requires the ability to detect and repair gaps in the stream, to provide a means to convey richer information about the nature of the data in context to what has come before and to enable the publisher to have control over who sees what.
In addition, a pubsub component should be able to provide information and parameters about its implementation of JRPS to subscribers. Subscribers must inquire about such information from the pubsub component to gain the full benefit of a JRPS service.

1. Identify the tasks that users can’t complete because we are lacking this crucial piece of protocol. (Note: users are not just IM users, but any person, system, or application that could gain value from interacting with Jabber.)

2. Discuss other projects or protocols and how Jabber could interface with them because of your proposed protocol enhancement (e.g., XML-RPC, SOAP, DotGNU).

3. Compare Jabber to ”the competition” (other IM systems or other messaging protocols) and point out holes in the Jabber protocol that need to be filled in order to offer similar functionality.

4. Review the relevant history of thinking within the Jabber community.

JRPS is a layer on Jabber Publish-Subscribe (XEP0024, XEP0036) and should interoperate with them and support namespaces and topics. Included in this document is the capability for permission tokens. It is included as the author believes that such tokens should exist within the <publish> tag, being a means to identify data much as the namespace or topic does.
JRPS is different from other IM systems in that the publisher and pubsub components send out the data so that downstream entities can detect if problems occur. As a comparison, a sender in, say, MSN is told that the packet they sent cannot be delivered but in JRPS, the receiver knows that a packet or packets have not been delivered and can ask for retransmissions. The sender need not normally know about such events as the intermediate components can usually cater for it. Thus JRPS has a future in areas such as Multicasting, large distributed and

proxy-based environments where the end subscribers may be very remote from the publisher. Existing commercial middlewares provide such facilities and it is especially necessary when data is pushed between applications and may not have an obvious ”context” in the stream to data immediately before or after. Thus, JRPS may seem over-the-top for a chatroom world, but is a basic requirement for, say, distributing real-time process states, events or persistent, mutable data.

2 Gap Detection And Repair

This can be achieved by the use of packet sequence numbering and heartbeats whilst avoiding the necessity to positively ACK each packet.

2.1 Sequence Numbering

Multiple levels of sequence numbers are envisaged and will be used in different circumstances. Multiple levels allow a rapid repair of short ”transient” breaks whilst catering for longer breaks, recoveries and resynchronisations without placing too great a burden on either subscriber or pubsub component. This discussion explains the use of a dual sequence number environment: link and source.
Sequence numbers will be sent in each publish thus:

```
<iq type="set"
    to="myclient@server.net"
    from="pubsub.localhost">
  <query xmlns="jabber:iq:pubsub">
    <publish ns="data_topics"
        linkseq="57372"
        sourceseq="7547392"
        from="publisher.fromaplace">
    </publish>
    <publish ns="data_topics"
        linkseq="57373"
        sourceseq="44211"
        from="publisher.elsewhere">
    </publish>
  </query>
</iq>
```

The above shows sequence numbers placed in the <publish/> node or element. This is to abstract the publishing from any packet construction algorithms that may occur and thus allow a recovery to make use of network capacity as it sees fit and to interleave recovery and ongoing publishing data.
The subscriber stub is responsible for ordering information and detecting and repairing any

gaps to provide sequential data for consumption by the application, which should not concern itself with such issues.
The operation of LINK and SOURCE sequence numbers are described below.

2.2 Link Level Swquence Numbering

This will concern itself with data sent on each channel. A channel can be, but is not limited to the following:

- Socket connection between Subscriber (and/or resource within same) and the Jabber pubsub component.
- Multicast datastream sent from a pubsub component but shared amongst 0..n subscribers.

Each publish received should contain an incremental sequence number to the previous or Zero. Zero is used to reset (or resynchronise) the sequence numbering. Zero should not be used in the situation of sequence number wrapping/rollover, wherein the value1 should be used. Sequence numbering bit resolution should be ascertained by querying the pubsub component in an <iq/> before subscription requests are levied.
E.g., in a 16-bit sequence number resolution channel, the sequence numbers would run as follows
1, 2, 3, 65533, 65534, 65535, 1, 2,
For information on sequence number bit resolution, see section 4, Source Queries.

2.3 Source Level Sequence Numbering

This will indicate the sequence number of messages sent from the publisher to the pubsub component. Should a link be lost, timeout or other such eventuality where the context of link sequence number be lost (e.g. the pubsub component decides the subscriber has disappeared and discards context), the pubsub component is still in a position to re-filter and retransmit data cached locally or even refer back to its source to maintain integrity and temporal ordering of data to the subscriber.
To repair larger gaps, the pubsub component may provide the capability to request upwards to the source using the source sequence number, or the pubsub component may draw upon local or remote journaling services to repair the gap. The source sequence number seen by the subscriber may be the link level sequence number between publisher and pubsub component, may be the ultimate publisher sequence number or even an internal sequence number given to the incoming published data to the pubsub component on a per source basis.
The subscriber need not know how the source sequencing operates, only notify from when the link last gave a contiguous datastream.
One can now see that the pubsub component's conversation to the source is akin to that of a

subscriber to a pubsub component.

2.4 Gap Filling

When a subscriber detects a gap on its link, it can request for the data to be resent thus:

```
<iq
    type="get"
    id="plugthegap1"
    from="myclient@server.net"
    to="pubsub.localhost">
  <query xmlns="jabber:iq:pubsub">
    <gap linkfrom="56737" linkto="56739">
  </query>
</iq>
```

The values represent the missed link sequence numbers. For a gap of 1, the linkfrom and linkto are the same.
Should the pubsub have lost the link context and thus is unable to plug the gaps it will return an error <iq/> packet.
All is not lost. The subscriber has a last-ditch repair scenario by sending last-received source sequence numbers.

```
<iq
    type="get"
    id="plugthegap1"
    from="myclient@server.net"
    to="pubsub.localhost">
  <query xmlns="jabber:iq:pubsub">
    <gap ns="publisher.fromaplace" after="56737">
    <gap ns="publisher.elsewhere" after="211234">
  </query>
</iq>
```

Due to the non-contiguous nature of source sequence numbers from the subscriber point of view, the values sent must represent not the gap, but the last valid sequence number received. Each source may have a separate sequence number stream. This allows the pubsub component to manage and, if necessary, request gaps itself from the publisher to resynchronise the subscriber. The pubsub or publishing source should have the ability to refuse a rebuild/resynchronise.
It should be possible for the subscriber to send the link and source sequence numbers in the initial request. However, if link information has been discarded by the pubsub component (e.g. the connection was dropped and presence set offline) the link sequence numbers will be reset to zero (re-synchronised) thus:

```
<iq
    type="set"
    to="myclient@server.net"
    from="pubsub.localhost">
<query xmlns="jabber:iq:pubsub">
    <publish
        ns="data_topics"
        linkseq="0"
        sourceseq="7547392"
        from="publisher.fromaplace">
    </publish>
    <publish
        ns="data_topics"
        linkseq="1"
        sourceseq="44211"
        from="publisher.elsewhere">
    </publish>
  </query>
</iq>
```

2.5 Heartbeats

During times of low traffic, an active circuit can be provided with regular heartbeat transmissions. Heartbeats will increment the link level sequence numbers. Subscribers missing or detecting overdue heartbeats will thus be able to detect gaps or delays even in low traffic scenarios. If the data is simply delayed, the subscriber stub is in a position to take action (and/or alert the application/user). If data is lost or heartbeats do not arrive in time, the subscriber can decide to request retransmission, disconnect or wait.

```
<iq
    type="set"
    to="myclient@server.net"
    from="pubsub.localhost">
  <query xmlns="jabber:iq:pubsub">
    <publish
        ns="link.heartbeat"
        linkseq="57374"
        from="pubsub.localhost">
    </publish>
  </query>
</iq>
```

No source sequence numbering exists here, as it is purely a link-level entity.

3 Publish Types

To be able to interpret published data in a more logical manner, more meaning needs to be given to data received.
When a publish packet arrives with a topic or data namespace, there is currently no way of knowing how to interpret the tags therein. Do they replace existing tag values seen? Should previously sent tags that are not in the publish be kept or discarded? Are tag values being updated or was the previous value incorrect?
To resolve this a type field may be added to the <publish/> tag.

```
<iq
    type="set"
    to="myclient@server.net"
    from="pubsub.localhost">
<query xmlns="jabber:iq:pubsub">
    <publish
        ns="data_topics"
        linkseq="57372"
        sourceseq="7547392"
        from="publisher.fromaplace"
        type="update">
    </publish>
    <publish
        ns="data_topics"
        linkseq="57373"
        sourceseq="44211"
        from="publisher.elsewhere"
        type="correction">
    </publish>
</query>
</iq>
```

This option is preferable to extending the <iq/> type field as there will then be no need to split <iq/> packets if <publish/> elements have different types.

3.1 Publish Type Field Values

The following extensions would be used in environments where topic/namespaces define discrete sets of data items and/or data items changing over time, as opposed to only referring to a topic datastream consisting of atomic, unrelated data. Other types can be defined as the need arises.
'update' - partial update of data. Replaces the values of the fields of the topic/namespace it contains. Other fields held/cached downstream for this data item are still valid.
'correction' - previous data for contained fields was incorrect - e.g. paragraph in a news story, but, as per update, unsent items are still valid.

'image' - payload contains ALL the data for a data item/topic/namespace. All existing values should be dropped and replaced with the new data. Previously received fields not now contained within the image should be discarded.
'drop' - namespace/topic item is now dead and all data in it should be deleted and purged from cache.
'snapshot' - requested by subscriber and is a request for data (an image if empty of granular topics/namespaces) and no further updates, as distinct from a get, which is an on-going subscription in pubsub world.
'add' - new topic/data item on publisher's feed. Note that an "image" publish for an item can be interpreted in the same way. Previous systems have had the ADD mechanism, but use of "add" has been discontinued, with the role taken up by the "image". (thoughts?)
The above states (except "add") are very important for downstream caches and for applications that apply business logic to the datastreams.

4 Source Queries

As touched on above, subscribers should be able to enquire of the publisher regarding what capabilities it provides and what to expect. Some items of use for JRPS are as follows:

- Heartbeat. Integer milliseconds. Represents the interval between heartbeat messages. Useful for rapid detection of link level problems.
- Link Sequence Resolution. Integer. Allows subscriber to predict link sequence number rollover. Zero would indicate that it is not supported.
- Source Sequence Resolution. Integer. Allows a subscriber to predict source sequence number rollover. A zero would indicate that it is not supported.
- Gap Support. Boolean. Used to indicate if the publisher supports gap filling.
- Permissioning Scheme. There may be some standardisation of permissioning schemes so that common plugins or mechanisms can be adopted. The publisher should be able to define this.
- Rebuild On Demand. If the pubsub can support a rebuild/refresh of the current values of all subscribed-to data on demand.
- Refresh Cycle. It has been common practice to transmit a regular refresh cycle for all subscribed data. If a data item does not get an 'image' from the source for a period of time, the cache performs a logical 'drop'. Without this, intermediate caches would very soon balloon with stale data, or publishers would get every cache re-requesting or confirming if data is still alive. Zero would indicate that no refresh cycle exists for the source.

5 Implementation Issues

5.1 Permissioning Requirements

Permissioning protocols should be open to permit a multitude of permissioning schemas. Data providers may wish to enforce their schemes in ways that suit their particular business models. The protocol should not bind or dictate such mechanisms.
The implementation of permissioning systems and regimes over time has repeatedly shown that it is especially dangerous to assume the behaviour of data and to disregard how information is used, protected, valued and owned or to force a scheme that is rigid and assumes a narrow problem domain. Thus the scheme should permit explicit and tokenised permissioning mechanisms.
Tokenised permissioning allows sets of data can be treated en masse. By permitting the concept of "grant" and "deny" permissions simultaneously (settings that define who CAN see something or defining who CANNOT) individual publishers can manage access both broadly and down to very fine granularity.
Permission tokens, if used, should be sent in-band with the data. This will allow data to change its coding online and thus immediately affect permissioning without a redistribution of the permissioning information.

```
<iq
    type="set"
    to="myclient@server.net"
    from="pubsub.localhost">
  <query xmlns="jabber:iq:pubsub">
    <publish
        ns="data_topics"
        type="update"
        permtoken="6747"
        linkseq="57372"
        sourceseq="7547392"
        from="publisher.fromaplace">
    </publish>
  </query>
</iq>
```

This does not prevent namespace/topic permissioning systems from being applied, nor should the permtoken be compulsory1.
The pubsub systems should be able to <iq/> the publisher for the permissioning regime that applies.
The definition of the XML carrying permissioning tables/information should be regime specific.
Further information on why tokenised grant and deny permissioning is advantageous can be provided upon request.

5.2 Contributions

Contributions in this context are when a subscriber publishes to one or more sources for redistribution so that it may reach the communities that subscribe to that source. By doing this, the subscriber reaches large communities, focus on specific communities and can abstract itself from delivery issues. The publisher gains information and broadens its appeal. Delivery abstraction is valuable, as a subscriber can then connect once to the publisher to gain access to all systems, networks, technologies, subscribers and media that the publisher and contributor agree upon. As you may guess, there is a need for content, flow control/throttling and ongoing permissioning to be specified and handled over time.
Contributions requires a separate XEP, but the issues are important to the implementation of pubsub and of its permissions (Contributors have specific, complex and business-critical reasons to tightly control who sees data -- e.g. only customers, not competition!)