Related commands

- PSUBSCRIBE
- PUBLISH
- PUBSUB
- PUNSUBSCRIBE
- SUBSCRIBE
- UNSUBSCRIBE

Pub/Sub

SUBSCRIBE, UNSUBSCRIBE and PUBLISH implement the Publish/Subscribe messaging paradigm where (citing Wikipedia) senders (publishers) are not programmed to send their messages to specific receivers (subscribers). Rather, published messages are characterized into channels, without knowledge of what (if any) subscribers there may be. Subscribers express interest in one or more channels, and only receive messages that are of interest, without knowledge of what (if any) publishers there are. This decoupling of publishers and subscribers can allow for greater scalability and a more dynamic network topology.

For instance in order to subscribe to channels `foo` and `bar` the client issues a SUBSCRIBE providing the names of the channels:

```
SUBSCRIBE foo bar
```

Messages sent by other clients to these channels will be pushed by Redis to all the subscribed clients.

A client subscribed to one or more channels should not issue commands, although it can subscribe and unsubscribe to and from other channels. The replies to subscription and unsubscription operations are sent in the form of messages, so that the client can just read a coherent stream of messages where the first element indicates the type of message. The commands that are allowed in the context of a subscribed client are SUBSCRIBE, PSUBSCRIBE, UNSUBSCRIBE, PUNSUBSCRIBE, PING and QUIT.

Please note that `redis-cli` will not accept any commands once in subscribed mode and can only quit the mode with `Ctrl-C`.

Format of pushed messages

A message is a Array reply with three elements.

The first element is the kind of message:

- `subscribe`: means that we successfully subscribed to the channel given as the second element in the reply. The third argument represents the number of channels we are currently subscribed to.
- `unsubscribe`: means that we successfully unsubscribed from the channel given as second element in the reply. The third argument represents the number of channels we are currently subscribed to. When the last argument is zero, we are no longer subscribed to any channel, and the client can issue any kind of Redis command as we are outside the Pub/Sub state.
- `message`: it is a message received as result of a PUBLISH command issued by another client. The second element is the name of the originating channel, and the third argument is the actual message payload.

Database & Scoping

Pub/Sub has no relation to the key space. It was made to not interfere with it on any level, including database numbers.

Publishing on db 10, will be heard by a subscriber on db 1.

If you need scoping of some kind, prefix the channels with the name of the environment (test, staging, production, ...).

Wire protocol example

```
SUBSCRIBE first second
*3
$9
subscribe
$5
first
:1
*3
$9
subscribe
$6
second
:2
```

At this point, from another client we issue a PUBLISH operation against the channel named `second`:

```
> PUBLISH second Hello
```

This is what the first client receives:

```
*3
$7
message
$6
second
$5
Hello
```

Now the client unsubscribes itself from all the channels using the UNSUBSCRIBE command without additional arguments:

```
UNSUBSCRIBE
*3
$11
unsubscribe
$6
second
:1
*3
$11
unsubscribe
$5
first
:0
```

Pattern-matching subscriptions

The Redis Pub/Sub implementation supports pattern matching. Clients may subscribe to glob-style patterns in order to receive all the messages sent to channel names matching a given pattern.

For instance:

```
PSUBSCRIBE news.*
```

Will receive all the messages sent to the channel `news.art.figurative`, `news.music.jazz`, etc. All the glob-style patterns are valid, so multiple wildcards are supported.

```
PUNSUBSCRIBE news.*
```

Will then unsubscribe the client from that pattern. No other subscriptions will be affected by this call.

Messages received as a result of pattern matching are sent in a different format:

- The type of the message is `pmessage`: it is a message received as result of a PUBLISH command issued by another client, matching a pattern-matching subscription. The second element is the original pattern matched, the third element is the name of the originating channel, and the last element the actual message payload.

Similarly to SUBSCRIBE and UNSUBSCRIBE, PSUBSCRIBE and PUNSUBSCRIBE commands are acknowledged by the system sending a message of type `psubscribe` and `punsubscribe` using the same format as the `subscribe` and `unsubscribe` message format.

Messages matching both a pattern and a channel subscription

A client may receive a single message multiple times if it's subscribed to multiple patterns matching a published message, or if it is subscribed to both patterns and channels matching the message. Like in the following example:

```
SUBSCRIBE foo
PSUBSCRIBE f*
```

In the above example, if a message is sent to channel `foo`, the client will receive two messages: one of type `message` and one of type `pmessage`.

The meaning of the subscription count with pattern matching

In `subscribe`, `unsubscribe`, `psubscribe` and `punsubscribe` message types, the last argument is the count of subscriptions still active. This number is actually the total number of channels and patterns the client is still subscribed to. So the client will exit the Pub/Sub

state only when this count drops to zero as a result of unsubscribing from all the channels and patterns.

Programming example

Pieter Noordhuis provided a great example using EventMachine and Redis to create a multi user high performance web chat.

Client library implementation hints

Because all the messages received contain the original subscription causing the message delivery (the channel in the case of message type, and the original pattern in the case of pmessage type) client libraries may bind the original subscription to callbacks (that can be anonymous functions, blocks, function pointers), using a hash table.

When a message is received an O(1) lookup can be done in order to deliver the message to the registered callback.
