```
BIP: 36
Layer: Peer Services
Title: Custom Services
Author: Stefan Thomas <justmoon@members.fsf.org>
Comments-Summary: No comments yet.
Comments-URI: https://github.com/bitcoin/bips/wiki/Comments:BIP-0036
Status: Rejected
Type: Standards Track
Created: 2012-08-03
License: PD
```

Abstract

This BIP adds new fields to the `version` message which clients can use to announce custom services without polluting the limited 64-bit `services` field. It also makes some non-binding recommendations regarding the implementation of custom services.

Motivation

We would like to encourage experimentation with custom services that extend the Bitcoin protocol with useful functionality. Examples include Distributed Hash Tables (DHT), distributed pools, lightweight client support protocols, directed message routing and support for custom transports. However, without a general framework for protocol extensions, these custom services are likely to collide in various ways. This BIP provides such a framework.

Specification

Two new fields are added to the `version` command, after `extra_height`:

Field Size	Description	Data type	Comments
1+	service_count	var_int	Number of extra services
?	service_list	service[]	List of service definitions

The service definitions `service[]` are given in the following format:

Field Size	Description	Data type	Comments
?	service_name	var_str	Unique service identifier
4	service_version	uint32_t	Identifies service version being used by the node
?	service_data	var_str	Additional service-specific data

A node MUST NOT announce two services with the same `service_name`. If a remote node sends such a `version` message the client MAY disconnect.

The `service_version` is service-specific and can be any integer. Higher versions SHOULD be higher integers. When a service is standardized, it is assigned a `NODE_*` constant for use with the `services` field and future iterations of the protocol depend on the Bitcoin protocol version. Both the `NODE_*` flag and the custom service entry MAY be provided for the duration of a transitional period.

Services SHOULD pass an empty string (0x00) as `service_data` and use a custom handshake to initialize their protocol, exchange information about capabilities etc. Note that to become a standardized service, a service MUST NOT rely on `service_data` since there is no corresponding mechanism for the standard services defined in the `services` field.

However, services MAY use `service_data` if they do not intend to become standard services and need a simple way to transmit a small amount of initialization data. For example, a node offering a custom transport like UDP or WebSocket, may choose to announce this as a service and include the port number in `service_data`. The format for `service_data` is service-specific and may be any binary or ASCII data. For ease of debugging, a human-readable (ASCII) format is generally recommended.

Service identifier

Each service SHOULD choose a new identifier that is not used by any other service. To register a new identifier, add it to the Service identifiers wiki page along with the name of the maintainer and a way to contact them. Please do not register identifiers unless you are actually using them.

Service identifiers that are reserved or used by an accepted BIP MUST NOT be used except in the way specified by that BIP.

Service identifiers MUST be between five (5) and eleven (11) characters long. Service identifiers MUST use only ASCII characters, excluding: / * _ :

Valid examples:

- `MySampleSvc`
- `smartserv`
- `P-Pool`

Valid, but discouraged examples:

- `MySVC 1.0` (use `service_version` to differentiate versions)
- `@@---.` (identifiers should be pronounceable)
- `lightweight` (avoid too generic names)

Invalid examples:

- `Pppc` (too short)
- `SuperService` (too long)
- `Cool_Svc` (invalid character)

Optional: Custom commands

Bitcoin command names are limited to 12 characters. That doesn't leave a lot of space for both the service identifier and the service command. Therefore we recommend that all service commands SHOULD be represented by a single "command" on the Bitcoin network. This command SHOULD consist of the exact service identifier to avoid collisions with other services, prefixed by an underscore to avoid collisions with current or future Bitcoin protocol messages. For example: `_MySampleSvc`

The service-specific command name SHOULD then be specified in an extra header in the payload:

Field Size	Description	Data type	Comments
12	subcommand	char[12]	ASCII string identifying the service command, NULL padded (non-N
?	subpayload	uchar[]	The actual data

The length of `subpayload` is derived from the length of the total payload minus twelve (12) bytes for the `subcommand`. Implementations MUST NOT rely on this format to be used by unknown services. Clients SHOULD ignore any services or subcommands they don't explicitly understand.

The recommended way to refer to messages following this format in documentation is by the service identifier, followed by a colon, followed by the subcommand. For example, the subcommand `search` for the `MySampleSvc` service would be referred to as: `MySampleSvc:search`

Full hexdump of an example `MySampleSvc:search` message:

```
0000   F9 BE B4 D9 5F 4D 79 53  61 6D 70 6C 65 53 76 63   ...._MySampleSvc
0010   14 00 00 00 73 D5 56 77  73 65 61 72 63 68 00 00   ....s.Vwsearch..
0020   00 00 00 00 12 34 56 78  9A BC DE F0               .....4Vx....

Message header:
 F9 BE B4 D9                                                           - Main networ
 5F 4D 79 53 61 6D 70 6C 65 53 76 63                                   - "_MySampleS
 14 00 00 00                                                           - Payload is
                                                                         (includes 1
 73 D5 56 77                                                           - Checksum

Service header:
 73 65 61 72 63 68 00 00 00 00 00 00                                   - "search" su

Search message:
 12 34 56 78 9A BC DE F0                                               - Payload
```

Standardization

Custom services may become standard parts of the protocol. Services which wish to become part of the Bitcoin protocol MUST fulfill the following criteria:

- MUST NOT use `service_data`; Standard services have no corresponding field
- MUST use a peer discovery mechanism which specifies one bit per node, same as the `services` field in `addr` messages
- MUST NOT use any subcommands that conflict with current or planned Bitcoin protocol commands

The standardization process will usually take place as follows:

1. The service is implemented and tested.
2. Once the API is known to be relatively stable it is formalized and submitted as a BIP.
3. Once the BIP is accepted, the service is assigned a `NODE_*` constant and the transitional period starts:
    - Clients MUST understand both the announcement of the service via the `services` field and via `service_list` and include both methods in their own `version` message.
    - Clients MUST accept both the wrapped form messages like `MySampleSvc:search` as well as the corresponding non-namespaced messages like `search`. Clients MUST only send wrapped messages.
    - During the transitional period the API of the service MUST NOT change.
4. After the transitional period:
    - Clients MUST only announce the service via the `services` field.
    - Clients MUST only send unwrapped messages.
5. Future changes to the service API now require a BIP and an increase in the Bitcoin protocol version.

This process of adding a service to the Bitcoin protocol should only be undertaken for services where there is a strong rationale for doing so. Services MAY also be standardized as custom services via a BIP while maintaining the custom service format.

Rationale

This BIP aims to fulfill the following goals:

- Minimize the risk of namespace collisions, ambiguities or other issues arising from conflicting custom services
- Provide an easy upgrade path for custom services to become standardized services with their own `NODE_*` flag
- Place minimum restrictions on custom service authors
- Allow custom services to be created with minimum effort
- Allow clients to support multiple/many custom services at once

To achieve these goals this BIP adds two new fields to the `version` message. It would have been possible to avoid changes to `version` by adding a new message instead. However, it makes sense to keep both types of service announcements in the same message so that the life cycle of standardized services and custom services remains exactly the same. This also simplifies detecting a service which is in the transition from a custom to a standardized service (and being announced using both methods.)

Finally, this BIP defines both explicitly and implicitly some useful common nomenclature that can be used when discussing custom services, e.g. "subcommand", "subpayload", "service identifier" and the colon format for referring to subcommands.

Copyright