Service Discovery specification

Documentation

- Service Discovery

Commands

Unsolicited Events

Events

Documentation

Service Discovery

Service Discovery

Introduction

Multiple services can be supplied by multiple vendors. This standard doesn't require coordination between these different organization, or between the service publishers and the service client. It is possible to operate a system with components from multiple hardware vendors, and with third party applications, without the prior knowledge of any party.

This specification covers an environment using WebSockets to communicate between services and applications, either on a single machine or across a network.

This section covers both the process for publishing a service such that it can be discovered, and the discovery process used by the service client.

There is also a clear definition of responsibility for each component, including when there are dependencies between components. There are no shared components required to coordinate the system.

The underlying network can use any protocol that supports WebSockets such as IPv4 or IPv6. Nothing in this document requires any particular underlying protocol.

This document uses CAN, WILL, MUST, SHOULD etc. in the normal ways to distinguish between requirements, recommendations, permissions and possibilities. See
https://www.iso.org/files/live/sites/isoorg/files/archive/pdf/en/how-to-write-standards.pdf

Requirements of this spec will be formated as follows:

> This is an example requirement

All other text is commentary used to illustrate the reasoning of the requirements. Where there is any conflict the requirement text always takes priority.

Overview

In this standard there are two types of "end-point"; publisher and service. Each end-point, of either type, is published by a single software/hardware vendor. A publisher end-point is used for service discovery, to discover service end-points. A single service end-point can expose multiple "services", where each service typically represents a single piece of hardware. A single machine (or a single IP address) may expose multiple publisher and service end-points from different vendors. A "client" application may consume multiple services from multiple service end-points on the same machine, or across multiple machines.

On startup of the machine, any software services attempt to claim access to individual network ports using the underlying operating system mechanism. Ports are claimed sequentially from a known sequence. Each port becomes an end-point that can publish multiple services from a single vendor.

A client application will attempt to connect to each port on a machine in the known sequence to get a list of all active publisher end-points. For each publisher end-point it then exchanges JSON messages across WebSockets with URIs using a known format to recover a list of services published by that end-point. Once it has a full list of services it can use WebSocket connections to communicate with each service to perform whichever actions are required.

Machine Identification

Machines publishing services are identified by URIs. Machines exposing end-points can be identified by an IP address or by a DNS name.

Either the IP address or DNS name for a machine must be known by the client for the client to connect. This would probably be a configuration setting for the application and would need to be known by the organization setting up the application, but this configuration is outside the scope of this document.

Network Protocol

TLS security will be used to secure network connections. The only exception will be when the network connection between the client and service can be physically secured because they are both inside the same cabinet. In that case it will be possible to use clear communication without TLS encryption.

> The publisher will publish all WebSocket services protected by TLS encryption. This will be identified by the wss:// protocol specifier.
>
> The publisher may publish WebSocket services without TLS encryption, as a clear WebSocket connection, but only if the physical connection between the service and the client is physically protected. It is up to the hardware manufaturer to ensure this physical protection is sufficient. This unsecured connection will be identified by the ws:// protocol specifier.
>
> Where TLS is used, the service will be protected by a mutually trusted server side certificate as part of the TLS protocol. This complete certificate chain must be mutually trusted by the client and service.

Establishing and managing the certificates between the service and the client is outside of the scope of this spec but trust must be in place. This might be achieved using a public third party certificate authority that issues TLS certificates. Alternatively it might be achieved using a bank's own internal CA. It shouldn't depend on a private CA or certificates issued by a vendor, which might limit access to the service.

> A WSS connections with invalid certificates will be invalid and will be rejected by both the client and the service.

URI Format

> Communication with service publishers and services will be through distinct URIs which will use the following format
>
> ```
> wss://machinename:portnumber/xfs4iot/v1.0/servicename
> ```

This consists of the following parts:

`wss://` **or** **`ws://`**
The protocol id for secure WebSockets. This should be wss:// for secure connections. An insecure ws:// connection can be used when the connection is physically secured, inside an ATM enclosure.

machinename
The identification of the machine publishing end-points. This can be an IP address or DNS name.

portnumber
The port number discovered through the initial service discovery process

XFS4IoT
A literal string. The inclusion of this part identifies standard XFS4IoT services published on this URI. It allows the possibility of a single vendor publishing standard and non-standard proprietary services on the same port. Any

standard service URI will start with this string. Any non-standard service's URI must not start with this string |

v1.0
The version of the XFS4IoT specification being used by this service. This will be updated in future versions of the specification and allows support for multiple versions of the specification on the same machine and end-point.

Note that most future changes to the XFS4IoT specification will be done in a non-breaking, backwards and forwards compatible way. For example, optional fields will be added to JSON messages when required. This means that changes to the version field of the URI will be very rare. It will only be changed if there is a breaking, incompatible change or a fundamental change to the API. Because of this there won't be any need for complex version negotiation between the client and the service. The client will simply attempt to open the version of the API that it supports.

ServiceName
This will be included in the URI to allow different services to be identified on the same port. Services will normally match individual devices. The exact service name is discovered during service discovery and is vendor dependent. The format of the service name shouldn't be assumed. The only URI that doesn't include a service name is the service discovery URI.

For example, a service discovery URI might be;

- wss://terminal321.atmnetwork.corporatenet:443/xfs4iot/v1.0
- wss://192.168.21.43:5848/xfs4iot/v1.0

Service URI might be;

- wss://terminal321.atmnetwork.corporatenet:443/xfs4iot/v1.0/maincashdispenser
- wss://192.168.21.43:5848/xfs4iot/v1.0/cardreader1

  The URI will be case sensitive. The URI will be lower case.

Service Publishing

Service publishers will negotiate access to resources and publish services using the following process.

Port Sequence

> Services will be published on a sequence of IP ports consisting of two ranges consisting of the port 80 and 443 followed by the ports 5846 to 5856 (inclusive.) Hence the full sequence of ports will be 12 ports as,
>
> 80 or 443, 5846, 5847, 5848, ... 5855, 5856
>
> Port 80 will only be used with HTTP/WS. Port 443 will only be used with HTTPS/WSS. All other ports may be used with either or both HTTP/WS and HTTPS/WSS.

Port 80 and 443 are the standard ports for HTTP and HTTPS and have the advantage that they are likely to be open on firewalls. The correct port will be used to match the protocol - 80 for HTTP/WS and 443 for HTTPS/WSS. Other ports are flexible and can be used for either protocol by the Service Publisher.

The port range 5846-5856 is semi-randomly selected in the 'user' range of the port space as defined by ICANN/IANA. This range is currently unassigned by IANA.

Note: *IP port ranges are conventionally assigned by ICANN/IANA so that multiple applications have the best chance of coexisting on the same IP address. However, there is no technical requirement for ports to be assigned and any process can attach to any port, with the risk of clashing with other services. Since financial hardware such as an ATM is unlikely to be running other services it's not actually critical that XFS4IoT ports are assigned by IANA.*

Even so, the XFS Committee should arrange to have this range assigned by IANA.

Free End-point Port Discovery

> On startup each service publisher must attempt to connect to the first port in the port sequence. It will

use the underlying OS and network stack to attempt to bind to this port.

All network access must go through the normal underlying OS mechanism. One service publisher must not block another publisher from accessing the network.

If the underlying OS reports that the port is already in use the service publisher will repeat the same process with the next port in the port sequence. This will be repeated until a port is successfully bound to, or all ports in the sequence have been tried.

If no available port can be found the service publisher will have failed to start. How this failure is handled by the service publisher is undefined.

It's important that a single organisation doesn't use up multiple ports, since this could lead to all the ports being blocks so that other publishers can't get a free port.

Any single organisation will publish all services on a single port, determined dynamically as above.

Note: *A service publisher will only fail to find a free port if more than 12 different hardware vendors are attempting to publish services from the same machine. This should be unusual.*

Handling Incoming Connections

Once a service publisher has successfully bound to a port it must handle connection attempts. It will accept all connections from any clients without filtering attempts. Security around connections will be handled after a connection has been established.

Note: *This document does not cover restrictions on connections to services or managing permissions for connections, such as limiting connections to certain machines or sub-nets. This would normally be under the control of the machine deployer and can be controlled through normal firewall settings and network configuration.*

Incoming connection attempts will specify a specific URI using the normal WebSocket process. The service publisher will allow connections to valid URIs as defined in this spec and track which URI each connection was made to.

The initial connection will be to the URI wss://machinename:port/xfs4iot/v1.0. This connection will then be used to list/discover individual services using the process outlined below (Service discovery).

Client

A client application must be able to discover and open a connection to each service that it will use. It does this in two steps; firstly, through publisher end-point discovery, then through service discovery for each service end-point. It will do this through the following process.

Publisher End-point Discovery

The client will enumerate end-points by attempting to open a WebSocket connection to the following URL on each port in the port sequence. (See Port sequence).
`wss://machinename:port/xfs4iot/v1.0`

The client will continue to enumerate publisher end-points by repeating for each port number in the port sequence until all ports have been tried.

The client will also start service discovery on the open connection. There is no requirement for the order of opening ports and discovering services. All ports connections may be created first followed by service discover, or port enumeration and service discovery may continue in parallel.

If the connection attempt to any port fails then the application will attempt error handling for network issues, machine powered off etc. The details of error handling are left up to the application.

Service End-point Discovery

Note: *This section uses message passing based on the standard command and message handling, specified elsewhere in this specificiation. In general there will be a Command/Response, Event, Completion, Unsolic.*

Once a connection has been established between the client and each publisher end-point, the client will discover the services published by sending a service discovery command and receiving events in the usual way. **To be defined when the messaging documentation is finalised.**

The only command sent to the publisher end-point will be "GetServices". This string is the literal command name.

The end-point will acknowledge the command in the normal way.

The command will be followed by zero or more events. The command will complete with a completion event, in the normal way. Each event, and the completion event will contain the following fields:

```
{
    "Command": "Services",
    "VendorName": "<Name of hardware/software vendor>",
    "Services":
    [
        {
            "ServiceURI": "wss://machinename:port/xfs4iot/v1.0/<servicename1>",
        },
        {
            "ServiceURI": "wss://machinename:port/xfs4iot/v1.0/<servicename2>",
        }
    ]
}
```

The service end-point URI will be returned as ServiceURI.

A secure wss:// protocol URI will be returned whenever possible.

An insecure ws:// protocol URI may be returned instead. If an insecure ws:// protocol is used then the hardware vendor will be responsible for ensuring the security of the connection.

Much of the security of the XFS4IoT specification is based around TLS encryption. Using an unencrypted ws:// protocol will have a negative impact on that security, so as far as possible a wss:// should always be used.

If an unencrypted ws:// connection is used then alternative methods should be used to keep the connection secure, prehaps by physically securing the connection.

The Publisher service will send an event to report on every URI. A single event may report on one or more URI. URI will not be repeated between events, so each URI will be reported exactly once.

A publisher service may be designed to send one URI per event, or it may group URI together into a smaller number of events. The publisher should try and send events to report on each URI as soon as each URI is known. It's possible a publisher will know the complete set of URI when they're requested and can send them all at once in one or more events. Alternatively the URI may not be known straight away (such as if an IP address or port is being dynamically allocated.) In that case the publisher service would delay sending events for unknown URI until the full URI is known.

Having each URI reported at most once means that a client can connect to each URI reported in events without having to track which URI have already been connected to. This simplifies the client. Alternatively, a client may wait for the completion event and a full set of URI before attempting to connect. This would be simpler to implement, but might be slower to start up.

The completion event will contain every URI that the publisher service is aware of.

The publisher service will follow the above process to publish all URI that it's aware of. It will not suppress URI based on device status or service status.

For example, a device might be powered off, in the process of powering on, or powered on but have a hardware fault that makes it impossible to use. In all cases the publisher service will publish the URI anyway. The client can't assume anything about the device based on the URI. It will always need to query the service at the URI for its status to know more.

Events should be sent as soon as a URI is known by the publisher - the event doesn't mean or imply that the URI is currently available or can be connected to - that error handling must be performed by the client (see below.)

Note: *Even if the publisher service could know that a URI was valid at the time that it sends the event, the client can't know that the URI is still valid when it attempts to use the URI. It could have failed between querying and connecting. So the client has to handle errors, timeouts and retrying when connecting to the URI.*

> The client may then attempt to open a WebSocket connection to each of the returned URI. The client will handle connection failures and timeouts by repeating the attempts to connect such that the service has a reasonable amount of time to start up.
>
> Each service will endeavor to accept connections as quickly as possible during startup and restarts. Once a connection has been accepted a service may continue to report a 'starting' status until the device is physically started and ready.

Some devices are slow physically to start up, but software should be able to start relatively quickly. So, for example, a cash recycler device might be able to accept a connection within a few seconds of power being applied, but the physical hardware can take several minutes to reset. During this time the service would accept connections but report a 'starting' status.

> Each connection will be used to communicate with a single service. The service will then be queried for details about that service, such as the type of service or device that it represents and the messages and interfaces that it supports. (**todo: Querying for service information needs to be documented elsewhere.**)
>
> The connection to the service will be kept open for as long as the service is in use. Details of the service lifetime are covered elsewhere.

The returned URI is a full URI including the machine name and port. It is possible that these values will be different to the service discovery URI - each service may be on a different machine, a different IP address, and a different port. The port is also independent of the discovery port range. It can be any port number.

> The service URI values will have the same version number as the service discovery URI version number. Different versions of the API will not be mixed.

If a client wants to open multiple different API version numbers then it should perform service discovery against each of the possible version URI strings.

> The client may close the publisher connection once it has completed service discovery, or it may keep the connection open. This will have no effect on the behavior of services.

XFS4IoT specification - Preview version 0.1. Initial limited stable release of this specification is expected Dec 2020. Next preview - Aug 2020. Note: work-in-progress. Subject to change without warning. Use at your own risk.

Note: *A future extension to the standard may cover dynamically adding and removing services, which will be reported by unsolic events from the Publisher service.*

XFS4IoT specification - Preview version 0.1. Initial limited stable release of this specification is expected Dec 2020. Next preview - Aug 2020. Note: work-in-progress. Subject to change without warning. Use at your own risk.

Commands

Unsolicited Events

Events