

**Introduction**

This section contains interleaved requirements and specifications.

A this stage this documentation is intended for internal consumption; sufficient enough to inform the prototyping and implementation phases

The requirements and specifications have been divided into the following sections

* [Architectural foundations](http://picosec.org:8080/specs/architecture)
* [Performance requirements](http://picosec.org:8080/specs/performance)
* [Naming and addressing (new section)](http://picosec.org:8080/specs/naming)
* [Raw connections and streaming](http://picosec.org:8080/specs/connections)
* [Service and device discovery](http://picosec.org:8080/specs/discovery)
* [Remote Service invocation](http://picosec.org:8080/specs/invocation)
* [Device administration](http://picosec.org:8080/specs/devadmin)
* [Enrolment and device binding](http://picosec.org:8080/specs/enrolment)
* [Secure connection](http://picosec.org:8080/specs/secure-connection) paddy
* [Policy definition](http://picosec.org:8080/specs/policy)
* [Data extraction (new section)](http://picosec.org:8080/specs/data)
* [Miscellaneous requirements (new section)](http://picosec.org:8080/specs/requirements)

# Architectural foundations

In this section we shall outline certain architectural foundations, background assumptions and technical terminology that underpins all other areas.

## IOT device

The IOT device is a typically small device, directly or indirectly connected to the internet that contains one or more sensors or actuators. The typical characteristics of an IOT device are

* has one or more attached sensors or actuators
* has an (intermittent) realtime connection, this connection may (but is not necessarily) and internet (IP) connection
* has low power consumption requirements
* has a small micro-controller capable of minor local processing

## IOT application

The IOT application is connected via internet protocols, either directly or indirectly, to the IOT device and operators on either the sensor streams, or controls the actuators.

## Driver vs Protocol Direct model

In typical IOT deployments there are two principle flavours of deployment architecture

* Driver model or Hub model assumes there is some intermediate entity between the IOT Device and the IOT application. This Hub device implements a driver, which converts between internet friendly IOT protocols to different (often) proprietary protocols and different bearers
* Protocol direct model: this supports the same protocol all the way from the IOT application to at least the micro-controller on the IOT device.

# IOT Protocol

We consider three application level protocols in scope for picosec.

* webinos
* CoAP
* MQTT

We will attempt to demonstrate both interoperability and security features across all three protocols

## webinos model

Webinos technology is based on JSON-RPC and feature URI feature discovery.

# CoAP Integration

we need to define how a CoAP session, and in particular the security aspects can interwork with CoAP

# MQTT Integration

we need to define how a MQTT session, and in particular the security aspects can interwork with MQTT

# Performance

The performance requirements parametrise certain key features on the underlying hardware

They are informed to a large extent form the hardware and radio analysis performed in the landscape analysis

## CPU requirements

Speed alone is an imperfect measure as the practical utility of the the microprocessor is a factor of the clock set and the the sophistication of the instruction set.

The prototypical CPU to be found in a IOT like system is represented by the TI 8051 family of chipsets

Microcontrollers Parametric MSC USB MHz 33 Up to 60 MIPS 8 Up to 30 Architecture 8-bit 8-bit Code Space Up to 32 kB Flash Up to 32 kB RAM RAM Up to 1.2 kB Up to 40 kB ADC Channels Up to 8 - ADC Resolution 24-bits - DAC Channels Up to 4 - DAC Resolution 16-bits - Vref Yes - Serial Interfaces SPI, I2C, USART USB, I2C, RS232, IrDA Pricing (1KU) $4.60 - $20.95 $1.15 - $5.10

This is a 33mhz chipset capability of processing 8 million instructions per second

For the purposes of picosec implementation we will consider 20MHz to be a minimal practical chip clock speed we will consider.

? 8MHz

## Memory requirements

Memory is required for:

* Code execution and long term data storage: typically flash RAM
* Working memory for executing code: SRAM

The security requirements for picosec raise the barrier higher than it typical for many IOT devices

We are setting our initial requirement at

1. FlashRAM: 64k
2. SRAM: 16k

Once we have have a working system, we many attempt to lower these initial requirements.

## Power requirements

Power is probably the most important facet of a working iot system.

Power determines how frequently a battery needs replacing. In many practical deployments the human costs of battery replacement, exceed the cost of the device, and in effect is the primary costs of the device.

We need to be looking at battery life of over 1 year.

A typical 9V battery has a storage of 450mAh <http://hwstartup.wordpress.com/2013/03/11/how-to-run-an-arduino-on-a-9v-battery-for-weeks-or-months/>

An typical AA alkaline battery has between 1800–2600 mAh <http://en.wikipedia.org/wiki/AA_battery>

“MCUs will have to consume less than 1µA and less than 200nA in deep sleep. Whatever ‘wins’ will have to offer 8bit type performance. - See more at:[http://www.newelectronics.co.uk/electronics-technology/the-internet-of-things-is-pushing-microcontroller-developers-to-move-in-unexpected-directions/59112/#sthash.AxzJikB0.dpuf](http://www.newelectronics.co.uk/electronics-technology/the-internet-of-things-is-pushing-microcontroller-developers-to-move-in-unexpected-directions/59112/" \l "sthash.AxzJikB0.dpuf)

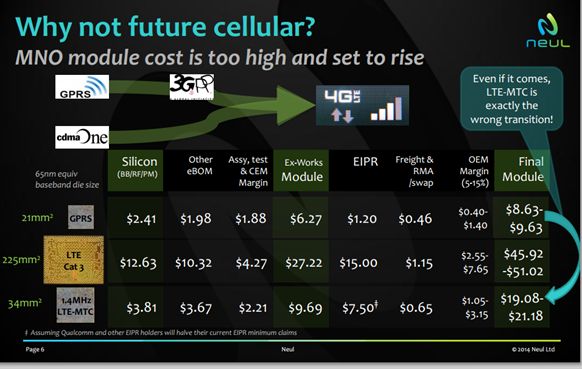
## Networking requirements

Putting aside the power consumption constraints of the radio technology, it is clear that an IOT device needs to connect.

Based upon our landscape analysis, we should consider all of these connections mechanisms in scope, accepting of course we do not have to implement all of them

* 802.11 - normal WIFI <http://en.wikipedia.org/wiki/IEEE_802.11>
* Bluetooth - <http://en.wikipedia.org/wiki/Bluetooth> with the following profiles
  + Serial <http://en.wikipedia.org/wiki/Bluetooth_profile#Serial_Port_Profile_.28SPP.29>
  + PAN <http://en.wikipedia.org/wiki/Bluetooth_profile#Personal_Area_Networking_Profile_.28PAN.29>
  + LE (Smart) <http://en.wikipedia.org/wiki/Bluetooth_low_energy>
* 802.15.4 - <http://en.wikipedia.org/wiki/IEEE_802>
  + ZigbeIP <http://www.zigbee.org/Specifications/ZigBeeIP/Overview.aspx>
  + 6lowpan <http://en.wikipedia.org/wiki/6LoWPAN>
* NFC
* SubGhz industrial, scientific and medical (ISM) radio bands <http://en.wikipedia.org/wiki/ISM_band>
  + (including 868 band)
* LAN bands
  + GSM <http://en.wikipedia.org/wiki/GSM>
  + 3G <http://en.wikipedia.org/wiki/3G>
  + HSPDA <http://en.wikipedia.org/wiki/High-Speed_Downlink_Packet_Access>
  + LTE <http://en.wikipedia.org/wiki/LTE_(telecommunication>)
* IOT optimised networks
  + Wightless <http://www.weightless.org/>
  + Sigfox <http://sigfox.com/en/>
* Satellite backhaul

One of the most significant characteristics of the networking technology is the price point

The following slide used by Neul demonstrates this quite effectively.

# Naming Schema

This schema should define the syntax of a URI that can be used to usefully describe the address of a target IOT device.

This URI should serve as a connection string

We should also consider that this same URI structure can also be used

* as a result of a service discovery process
* as a lexical item in a policy file (ie describing rights/restrictions) with respect to named entities

We should consider whether we have both a HTTP/S schema and another schema with different semantics

* e.g. HTTP - returns human readable descriptors and connection string lings
* picosec:// binds to a runtime that can make a connection

## Addressing/Naming requirements

REQ Each IOT device needs globally unique name (when fully qualified)

* This name should interoperated with existing internet technologies
  + Make URI a logical choice
* Using this name it should be possible to deterministically resolve to a physical address for that same device
* This name should be able to resolve in the following example scenarios
* Cloud roaming scenarios
  + A GPS sensor on phone should be resolvable on a mobile network
  + This same URI should still work when the phone roams from GSM network A to WIFI network at home
  + This same URI should still resolve to the same sensor when roans on to WIFI network at work

## NonIP resolving scenarios (same as paddys BUS attached)

* Should resolve to IOT device on Bluetooth serial
* Should resolve on subIP Zigbee profile 802.14.5
* Should resolve on general serial port connection
* Should resolve on subIP ne
* Should resolve over NFC?

The same physical device should be able to roam across/between all the connection methodologies above, but name should not change.

The addressing syntax should directly support the following elements

* Address of IOT owner
* Address of IOT host device (in scenarios where device proxies for sensor)
* Address of service instances With an option also to specific type of service e.g. phone supports two GPS instances (WIFI and GPS) – but both are of the same type

## Privacy Disclosure Implication

As discussed extensively at the W3C Web of Things workshop <http://www.w3.org/2014/02/wot/> URI representation of IOT devices is essential for web level interoperability, irrespective of the actually connection technology used.

If an IOT Object name is public, we must consider what if any private information is inadvertently disclosed to a recipient of the address.

A landline telephone number for example, as a portable address, discloses both country and region to the holder of that address.

We must consider carefully whether the hierarchical structure discloses similar types of information, and factor that into the design.

Illustratively we have two primary variants

* Structured hieriachy
  + <http://hostdomain.com/OwnerID/HubID/DeviceID/ServiceID>
  + this is a useful structure, identifying deterministically locale of the referenced device and service
  + howwever in the process id discloses the names of devices and servers in the hierarchy
* Anonymised token:
  + <http://hostdomain.com/sdf78sdasdf876sdf87s6d8a7f68as6d8f6as8d6fa8sd6f8a7sd6f8as>
  + a long unique key which dereferences to the item in question.

Which technique is preferred is determined to a large extent by how freely device identifiers will be shared

1. Will they be shared between trusted individuals?
2. OR will they be published on open directory services?

## Sharing sphere

When we move to consider the nature of an IOT connection, and the security qualities we expect from it, pragmatically it is important to consider how accessible, discoverable an object address is.

For example WPS security is based on a very simple challenge-response security mechanism. Essentially two buttons being pressed within a very short space of time with one another.

This is secure, because the physical characteristics of WIFI mean the universe of device which can attach (issue WPS challenge or response) is restricted to only those devices within a physically <100m.

Essentially there are two subsequent hurdles that need to be overcome to compromise an IOT device

1. Discovery physical address of device
2. Compromise connection security of device

If we increase 1 we can decrease 2, and vice versa

The point of this section is to underline the fact that the discoverability of device name is an integral part of the IOT device security consideration

# Proposed Naming Specification

# Physical connections

In scope we should consider the following physical bearers

* 802.11 - normal WIFI <http://en.wikipedia.org/wiki/IEEE_802.11>
* Bluetooth - <http://en.wikipedia.org/wiki/Bluetooth> with the following profiles
  + Serial <http://en.wikipedia.org/wiki/Bluetooth_profile#Serial_Port_Profile_.28SPP.29>
  + PAN <http://en.wikipedia.org/wiki/Bluetooth_profile#Personal_Area_Networking_Profile_.28PAN.29>
  + LE (Smart) <http://en.wikipedia.org/wiki/Bluetooth_low_energy>
* 802.15.4 - <http://en.wikipedia.org/wiki/IEEE_802>
  + ZigbeIP <http://www.zigbee.org/Specifications/ZigBeeIP/Overview.aspx>
  + 6lowpan <http://en.wikipedia.org/wiki/6LoWPAN>
* NFC
* SubGhz industrial, scientific and medical (ISM) radio bands <http://en.wikipedia.org/wiki/ISM_band>
  + (including 868 band)
* LAN bands
  + GSM <http://en.wikipedia.org/wiki/GSM>
  + 3G <http://en.wikipedia.org/wiki/3G>
  + HSPDA <http://en.wikipedia.org/wiki/High-Speed_Downlink_Packet_Access>
  + LTE <http://en.wikipedia.org/wiki/LTE_(telecommunication>)
* IOT optimised networks
  + Wightless <http://www.weightless.org/>
  + Sigfox <http://sigfox.com/en/>
* Satellite backhaul

## Logical connections

All of the above physical networks can be encapsulated by one or more of the following transport protcols

* TCP: <http://en.wikipedia.org/wiki/Transmission_Control_Protocol>
* UDP: <http://en.wikipedia.org/wiki/User_Datagram_Protocol>
* Serial: which is a lower layer simple read write protocol

## Session handling

Session handling will be dealt with in the security section

## Connection requirements

Requirement: A connection should be requested to a destination URI representing and IOT device – or vice versa –the IOT device connects to and end point

Requirement: The connection algorithm can resolve to the destination end point using its knowledge of the connection bearers available

Requirement: It should be possible to specific optional parameters to the connection, to indicate a bearer preference.

Requirement: The external application interface for sending and receiving data should be independent of bearer – work across TCP and Serial

Requirement: The external application interface for sending and receiving data should be accessible from JavaScript and C interfaces

Requirement: The external application interface for sending and receiving data should support frame based packets interface, in order to identify out of order and missing packet

Requirement: There should be an option to select a stream based interface to data, where ordering and resends are handled by application layer (but possibly standardised) logic – where the undelying bearer does not procide these functions

Requirement: Where the underlying bearer technology supports streaming, this should be made directly available to the application layer.

## Connectivity requirements

Requirement:Connections must be possible between organisations on a peer to peer basis, without the need for an intermediate entity

Requirement: Connections between entities be capable of resisting man in the middle attacks

Requirement: Connections between organisations must be mutually attestable

Requirement: Mutually authenticated TLS sessions are the preferred mechanism of creating a mutually attestable virtual connection

Requirement: Secure connections should be possible over non-IP connections

Requirement: End points must be named using general purpose URI

Requirement: It must be possible to route to end points, even when sitting behind firewall

Requirement: Virtual connections must have a notion of session longer than physical sessions

## Identity/Integrity/Session requirements

Requirement: When two devices connect there should be a handshake process in order that they may reliably assure each other of their identity

Requirement: This handshake process should produce a session ID which can be sent with all subsequent messages

Requirement: This session ID should be passed in such a way that it cannot be spoofed by sender

Requirement: New session IDs (requiring re handshake) should be requestable at any time by either party

There should be default behaviours defined regarding retirement of session ID

Requirement: These requirements should be embodies with the external Connection request and data transfer APIs

Requirement: Where the underlying bearer supports TLS – there should be an option to implement all these requirements directly on top of TLS using mutually authenticated sessions

## Reliability requirement

The fundamental difference between TCP and UDP/Serial based communication is the reliability of the data packets sent. In TCP we have

1. Guaranteed delivery: packets are explicitly acknowledged so we know data gets there
2. Ordered delivery: packets come in the correct order

UDP and Serial communication do not have these assurances.

Some IOT applications do not need this requirement.

The streaming API to be developed needs to have an option for unreliable delivery.

## Connection qualities

We identify thee orthogonal characteristics of connection types to consider in an IOT devices

* Reliable
* Non Spoofable
* Confidential

### Reliable connections

Reliable connections have two essential characteristics

* acknowledged delivery: a sender of a message knows that the messages has been received
* ordered delivery: packets sent by sender are guaranteed to be recieved in order by the recipient; no out of order messages

### Non-Spoofable

This relates to the quality of the connection that the receiver has a high degree of assurance that the sender of the message is who he says he his.

For our purposes this relates to the device identifier, or more correctly an identity given to that physical device.

These qualities are variously known as:

* device identity
* cryptographic device identity
* immutable (cryptographic) identity
* and/or hardware-rooted/hardware-protected key

This quality needs to apply in the other direction also: the sender having confidence in the identity of the recipient

### Confidentiality

Simply is the quality of the message being encrypted so that it cannot be snooped.

## Connection use cases

Using these othogonal connection qualities it is possible to outline eight different connection types, each represented in a different cell of two 2x2 matrices

When we consider these use cases we need to be pragmatic evaluating the business context. It is easy to give a theoretical answer that ALL connections must have the highest security credential.

However if we look at modern day pervasive and even business critical communications it is clear that

* Email
  + Unreliable
  + Spoofable
  + Non confidential
* SMS
  + Unreliable
  + Spoofiable
  + Confidential

### Unreliable

Unreliable IOT use cases cover the following scenarios

* Audio streaming
* Video streaming
* Sensor sampling (where sample loss can be tolerated)

|  |  |  |
| --- | --- | --- |
|  | Spoofable | Non Spoofable |
| Non Confidential | Coffee webcam Weather cam Commercial energy monitor | NULL |
| Confidential | Traffic cam TFL bus locations | Baby cam Car insurance black box Domestic energy |

### Reliable

Reliable streams cover the following classes of usecases

* Actuator commands
* Device configuration
* Metering events - generating charge
* Mission critical events

|  |  |  |
| --- | --- | --- |
|  | Spoofable | Non Spoofable |
| Non Confidential | NULL | NULL |
| Confidential | Domestic energy usage Smoke detector | Energy billing |

# Discovery

Discovery of IOT devices needs to be considered at two levels

* Device discovery
* Service discovery

## Device discovery

This is the discovery of a physical device which may be accessible over one or more networks. Each device is uniquely named, and uniquely referenceable, enabling connections to devices to be restablished

## Service discovery

Each device can host one or more services. Similarly each service needs to be uniquely referenceable.

Discovered services should also be typed. For example a GeoLocation type of service may have multiple implementations on a device.

## External discovery integration

A service, when implemented directly on a device, or indirectly via a driver should be capable of advertising itself and/or responding to service discovery requests made upon the device.

There exist a number of “other” technologies which also implement service description. A well designed device and service discovery mechanism should integrate with existing schemes.

This includes:

* mDNS: for physical device discovery <http://en.wikipedia.org/wiki/Multicast_DNS>
* DNS-SD: service discovery mechanism
* UPnP: for more advanced service discovery mechanism
* Bluetooth: service discovery
* CoAP: using content type http headers mechanmsim

## Discovery Requirements

A simple protocol should be provided – which only runs on top of “stream” interface for discovering available services

Advertised services should be published by using a scheme similar to the URI addressing schema

## API requirements

### General API requirements

Requirement: Definition: an API is a collection of methods, events and data definitions that addresses a functional area Requirement: Each API must be mapped to an interoperable JavaScript interface formally specified Requirement: Each API must be identified by a unique type that can issued on a distributed basis – there is no central authority for API names Requirement: Each API type should be instantiable multiple times on a particular host device Requirement: Each API instance should be given a unique name to identify it Requirement: All APIs should share common conventions for return codes, errors and parameter passing Requirement: APIs should distinguish between errors of function and errors of connectivity Requirement: The external interface to the solution must be via open, non-proprietary interfaces.

### Remotable API requirements

Requirement: The existence of an instance of an API should be discoverable by a remote device Requirement: An API should be invokable by a remote device. Requirement: A discovery protocols must exist to allow entity to discover data and services resident on each other’s organisations. Requirement: The solution must be able to remotely list the List APIs or services hosted on a server

### Wildcard Searching

Requirement: it must be possible to search for services without specifying device, in which case there must be a universe of devices implied by the search

Requirement: it must be possible to search for devices without specifying service or service-type

Requirement: it must be possible to search either a device or (implied) device set to search on device type.

## CoAP-webinos discovery alignment

TBD

# Discovery Specification Draft

Discovery needs to be accessible via API, whether by C-like native API or webfriendly JavaScript.

## Discovery API

The following draft specification is informed by the webinos discovery API and encapsulated in JavaScript

[Discovery API](http://picosec.org:8080/specs/apis/servicediscovery.html)

## Discovery Protocol

The service discovery messaging is based on messaging specified in invocation section. The Service Discovery API relies on messages that are sent on the session layer level to exchange information about registered services between the IOT-Device and connected IOT applications. For different implementations to be compatible these messages must be supported. The general session layer message format is the following:

{

"to" : to, // the receiver

"from" : from, // the sender

"type" : "prop", // must be "prop"

"payload" : {

"status" : "", // a string defining the message type, see below

"message" : {} // the body depending on status

}

}

The Service Discovery uses three types of messages of that format. The following specifies how the payload property of the above object should be formatted for each of them.

### registerServices

This message is used by a iot-device to inform the iot-app of its own registered services. It is used initially when the iot-device connects to the iot-app and every time there is a change in the registered services with the iot-device.

{

"status" : "registerServices",

"message" : {

services : [], // an array of service objects according to the Service interface from the Service Discovery API specification

from: "" // a string representing the iot-device id

}

}

### findServices

This message is a challenge sent every time a iot-device asks a iot-app or iot-device for their services. The receiver of this message should then reply with the @foundServices@ message specified below.

{

"status" : "findServices",

"message" : {

id: "" // an id that must be echoed in the reply (foundServices, see below) to identify the callback

}

}

### foundServices

This message is sent as reply to the @findServices@ message and includes all services from the sender.

{

"status" : "foundServices",

"message" : [] // an array of service objects according to the Service interface from the Service Discovery API specification

"message.id" : "" // the same id that was received from the findServices message that triggered this message as response

}

}

## Service Naming

A service is named by a FEATURE-URI

A feature URI a string which represents the type of a service.

As a URI it can be specified by any party. The assumption being the owner of the domain in the URI is the owner of the service definition.

Past the domain identifier there is no prescribed format. The domain owner is free to define their own identifies.

There is one optional parameters that can be specified as request parameters on the URI

**version** a number represented by string

## Service Description

Services are described by WebIDL

<http://www.w3.org/TR/WebIDL/>

WebIDL can be bound to C++ code using the Mozilla schema

<https://developer.mozilla.org/en-US/docs/Mozilla/WebIDL_bindings>

or indeed the Chromium conventions

<https://code.google.com/p/es-operating-system/wiki/CplusplusBinding>

(these need comparing)



# [picosec developer's documentation](http://picosec.org:8080/)

Other Resources

The picosec Discovery API provides web applications with an API to discover services without any previous knowledge of the service.

* [Introduction](http://picosec.org:8080/specs/apis/servicediscovery.html#intro)
* [Interfaces and Dictionaries](http://picosec.org:8080/specs/apis/servicediscovery.html#interfaces)
  + [DiscoveryInterface](http://picosec.org:8080/specs/apis/servicediscovery.html#::DiscoveryInterface)
  + [ServiceType](http://picosec.org:8080/specs/apis/servicediscovery.html#::ServiceType)
  + [FindCallBack](http://picosec.org:8080/specs/apis/servicediscovery.html#::FindCallBack)
  + [Service](http://picosec.org:8080/specs/apis/servicediscovery.html#::Service)
  + [BindCallBack](http://picosec.org:8080/specs/apis/servicediscovery.html#::BindCallBack)
  + [Options](http://picosec.org:8080/specs/apis/servicediscovery.html#::Options)
  + [Filter](http://picosec.org:8080/specs/apis/servicediscovery.html#::Filter)
  + [ServiceLocation](http://picosec.org:8080/specs/apis/servicediscovery.html#::ServiceLocation)
  + [PendingOperation](http://picosec.org:8080/specs/apis/servicediscovery.html#::PendingOperation)
  + [picosec](http://picosec.org:8080/specs/apis/servicediscovery.html#::picosec)
* [Enums](http://picosec.org:8080/specs/apis/servicediscovery.html#enums)
  + [ServiceState](http://picosec.org:8080/specs/apis/servicediscovery.html#::ServiceState)
* [Features](http://picosec.org:8080/specs/apis/servicediscovery.html#api-features)
* [Full WebIDL](http://picosec.org:8080/specs/apis/servicediscovery.html#full-webidl)

# Discovery API

## Authors

© 2014 [picosec consortium](http://www.picosec.org/).

## Summary of Methods

| **Interface** | **Method** |
| --- | --- |
| [DiscoveryInterface](http://picosec.org:8080/specs/apis/servicediscovery.html#::DiscoveryInterface) | [PendingOperation](http://picosec.org:8080/specs/apis/servicediscovery.html#::PendingOperation) [findServices](http://picosec.org:8080/specs/apis/servicediscovery.html#findServicesid134000)(ServiceType serviceType, FindCallBack findCallBack, Options? options, Filter? filter) DOMString [getServiceId](http://picosec.org:8080/specs/apis/servicediscovery.html#getServiceIdid134188)(DOMString serviceType) |
| [FindCallBack](http://picosec.org:8080/specs/apis/servicediscovery.html#::FindCallBack) | void [onFound](http://picosec.org:8080/specs/apis/servicediscovery.html#onFoundid133309)(Service service) void [onLost](http://picosec.org:8080/specs/apis/servicediscovery.html#onLostid133822)(Service service) void [onError](http://picosec.org:8080/specs/apis/servicediscovery.html#onErrorid133673)(DOMError error) |
| [Service](http://picosec.org:8080/specs/apis/servicediscovery.html#::Service) | void [bindService](http://picosec.org:8080/specs/apis/servicediscovery.html#bindServiceid133026)(BindCallBack bindCallBack, DOMString serviceId) void [unbindService](http://picosec.org:8080/specs/apis/servicediscovery.html#unbindServiceid132465)() |
| [BindCallBack](http://picosec.org:8080/specs/apis/servicediscovery.html#::BindCallBack) | void [onBind](http://picosec.org:8080/specs/apis/servicediscovery.html#onBindid132413)(Service service) void [onUnbind](http://picosec.org:8080/specs/apis/servicediscovery.html#onUnbindid131657)(Service service) void [onServiceAvailable](http://picosec.org:8080/specs/apis/servicediscovery.html#onServiceAvailableid131718)(Service service) void [onServiceUnavailable](http://picosec.org:8080/specs/apis/servicediscovery.html#onServiceUnavailableid131832)(Service service) void [onError](http://picosec.org:8080/specs/apis/servicediscovery.html#onErrorid131491)(DOMError error) |
| [PendingOperation](http://picosec.org:8080/specs/apis/servicediscovery.html#::PendingOperation) | void [cancel](http://picosec.org:8080/specs/apis/servicediscovery.html#cancelid143174)() |
| [picosec](http://picosec.org:8080/specs/apis/servicediscovery.html#::picosec) |  |

## Introduction

The Discovery API is not limited to discovery of local services but also enables discovery of remote services.

The API enables discovery of services that is exposed either:  
  
1. in the device  
2. by entities directly connected to the device,  
3. by entities available on the same local IP network  
4. by trusted services registered in a personal zone.   
  
Once a service is found the API will provide a service object that is used to bind to a service and monitor the availability of the service. The binding to a service will make sure that the user is authorized to use the service, create an implementation of the API and establish a communication path to the remote peer providing the service. The service object hides the complexity of communicating over different bearers, do cross network addressing, traversing NAT/Firewalls and connection management.

Prerequisites for the Discovery API is that the web application using the API is installed, trusted and that the user of of the device is authenticated and authorized to use the API.

## Interfaces and Dictionaries

### Interface DiscoveryInterface

The DiscoveryInterface interface provides functionality for discovery of services. The API supports the possibility to discover services based on a given service type either in a personal zone of trusted services or via other legacy discovery methods such as Bluetooth SD, DNS SD, mDNS or UPnP. When searching for a service type the operation can be restricted by providing certain constraints and/or context information via a filter interface.

##### WebIDL

        [NoInterfaceObject] interface DiscoveryInterface  {  
  
  
  
                 [PendingOperation](http://picosec.org:8080/specs/apis/servicediscovery.html#::PendingOperation) findServices(    
  
                        [ServiceType](http://picosec.org:8080/specs/apis/servicediscovery.html#::ServiceType) serviceType,   
  
                        [FindCallBack](http://picosec.org:8080/specs/apis/servicediscovery.html#::FindCallBack) findCallBack,   
  
                        optional [Options](http://picosec.org:8080/specs/apis/servicediscovery.html#::Options)? options,   
  
                        optional [Filter](http://picosec.org:8080/specs/apis/servicediscovery.html#::Filter)? filter);  
  
                   
  
  
  
                 DOMString getServiceId(DOMString serviceType);  
  
                   
  
        };

The code example below shows how an application initiates a search query to find a geolocation service. Whenever a service is found, a new HTML selection item is added to an HTML option list. Once the user selects a service, the usage of the service is authorized and an implementation of the API is instantiated by binding to the service.

##### Code example

        var findHandle = 0;  
        var serviceHandle = 0;  
        var geoServices = {};  
        var serviceId;  
          
        // Callback that displays a list of found services in a HTML selection list  
        // The selection list is dynamically extended every time a new service is discovered.     
        function serviceFoundCB(service) {        
                var selectlist = document.getElementById('servicelist');  
                var option = document.createElement('option');  
          
                option.value = service.id;  
                option.id = service.id;  
                option.appendChild(document.createTextNode(service.displayName));  
                geoServices[service.id] = service;  
                selectlist.appendChild(option);  
        }  
          
        // Callback that removes a service from the selection list of found services,  
        // when the service is not available any longer.   
        function serviceLostCB(service) {         
                var option = document.getElementById(service.id);  
          
                geoServices[service.id] = NULL;  
                option.parentNode.removeChild(option);  
        }  
          
        // Success callback when bindService has been successfully executed on the service object.   
        function bindCB(myLocationService) {  
                alert('Service ' + myLocationService.displayName + ' ready to use');  
                myLocationService.getCurrentPosition(showMap);  
        }  
          
        // Event listner that is called when the 'change' event is dispatched on the HTML selection list.    
        function serviceSelected(service) {       
                // Stops the findServices operation  
                findHandle.cancel();  
          
                // Binds to the service to initiate an authorized object used to  
                // invoke services.    
                service.bindService({onBind:bindCB});  
        }  
          
        if (serviceId) {  
                // If serviceId is known, bind to the service directly. Assumes  
                // that serviceId is stored persistently or received via an out  
                // of band channel.   
                  
                window.picosec.discovery.findServices(  
                        {api:'http://picosec.org/api/w3c/geolocation'},   
                        {onFound:serviceFoundCB, onLost:serviceLostCB},  
                        null,  
                        {serviceID : serviceId});  
                  
                serviceHandle = window.picosec.discovery.findServices(  
                        {api:'http://picosec.org/api/w3c/geolocation'},   
                        {onFound:serviceFoundCB, onLost:serviceLostCB},  
                        null,  
                        {serviceID: serviceId});  
        }   
        else {  
                // Initiate a search query for a service of the type geolocation  
                findHandle = window.picosec.discovery.findServices(  
                        {api:'http://picosec.org/api/w3c/geolocation'},   
                        {onFound:serviceFoundCB, onLost:serviceLostCB});  
          
                var selectlist = document.getElementById('servicelist');  
                selectlist.addEventListener("change", function (e) {  
                        var service = geoServices[e.target.value];  
                        if (service) {  
                                serviceSelected(service);  
                        }  
                }, false);        
        }

#### Methods

**findServices**

The findServices method initiates an asynchronous search query for services matching the requested serviceType and filter parameter. The method continues to search for services until the findServices method is canceled by the application or when the maximum search timer expires. The zones in which services are to be searched are expected to be managable by the picosec runtime.

##### Signature

[PendingOperation](http://picosec.org:8080/specs/apis/servicediscovery.html#::PendingOperation) findServices([ServiceType](http://picosec.org:8080/specs/apis/servicediscovery.html" \l "::ServiceType) serviceType, [FindCallBack](http://picosec.org:8080/specs/apis/servicediscovery.html#::FindCallBack) findCallBack, optional [Options](http://picosec.org:8080/specs/apis/servicediscovery.html#::Options)? options, optional [Filter](http://picosec.org:8080/specs/apis/servicediscovery.html#::Filter)? filter);

##### Parameters

* **serviceType**
  + **Optional:**No.
  + **Nullable**: No
  + **Type:**[ServiceType](http://picosec.org:8080/specs/apis/servicediscovery.html#::ServiceType)
  + **Description:**An input argument that defines which type of API that is requested. The serviceType is an URI that uniquely identifies the API.
* **findCallBack**
  + **Optional:**No.
  + **Nullable**: No
  + **Type:**[FindCallBack](http://picosec.org:8080/specs/apis/servicediscovery.html#::FindCallBack)
  + **Description:**Callback interface used to report the outcome of the search process.
* **options**
  + **Optional:**Yes.
  + **Nullable**: Yes
  + **Type:**[Options](http://picosec.org:8080/specs/apis/servicediscovery.html#::Options)
  + **Description:**Defines search options.
* **filter**
  + **Optional:**Yes.
  + **Nullable**: Yes
  + **Type:**[Filter](http://picosec.org:8080/specs/apis/servicediscovery.html#::Filter)
  + **Description:**Defines a filter that be used to limit the service operation to certain constraints and context information.

**getServiceId**

For the case that the application itself exposes functionality via JavaScript service interfaces the getServiceId method generates a service identity that can be shared with other peers to establish a binding without invoking a findServices operation. If no matching API is found the method will return Null.

##### Signature

DOMString getServiceId(DOMString serviceType);

##### Parameters

* **serviceType**
  + **Optional:**No.
  + **Nullable**: No
  + **Type:**DOMString
  + **Description:**URI identifying which type of API that shall be exposed. The URI shall be must be declared in the manifest in the api-name attribute of the picosec:shared-api element.

### Dictionary ServiceType

The Service Type dictionary is used to define which type of service that is requested.

##### WebIDL

        dictionary ServiceType {  
  
  
  
                DOMString api;  
  
        };

#### Dictionary Members

**DOMString api**

URI used to identify which type of API that is requested. The URI could either be:  
  
1. picosec Feature URI that is defined for each API by picosec, for example http://picosec.org/api/sensors/temperature,  
2. picosec Feature URI for referred unchanged API specifications, for example http://picosec.org/api/w3c/geolocation,  
3. an unique URI identifying an API exposed by a web application. The URI shall be the same URI as exposed by the web application manifest in the api-name attribute of the picosec:shared-api element.

### Interface FindCallBack

FindCallBack interface definition

##### WebIDL

        callback interface FindCallBack {  
  
  
  
                 void onFound([Service](http://picosec.org:8080/specs/apis/servicediscovery.html#::Service) service);          
  
                   
  
  
  
                 void onLost([Service](http://picosec.org:8080/specs/apis/servicediscovery.html#::Service) service);  
  
                            
  
  
  
                 void onError(DOMError error);  
  
                                   
  
        };

#### Methods

**onFound**

Asynchronous callback used whenever a new service is found.

##### Signature

void onFound([Service](http://picosec.org:8080/specs/apis/servicediscovery.html#::Service) service);

##### Parameters

* **service**
  + **Optional:**No.
  + **Nullable**: No
  + **Type:**[Service](http://picosec.org:8080/specs/apis/servicediscovery.html#::Service)
  + **Description:**An input argument representing the found service.

**onLost**

Asynchronous callback used whenever the state of a service change from available to unavailable.

##### Signature

void onLost([Service](http://picosec.org:8080/specs/apis/servicediscovery.html#::Service) service);

##### Parameters

* **service**
  + **Optional:**No.
  + **Nullable**: No
  + **Type:**[Service](http://picosec.org:8080/specs/apis/servicediscovery.html#::Service)
  + **Description:**An input argument representing the lost service.

**onError**

Asynchronous error callback.

##### Signature

void onError(DOMError error);

##### Parameters

* **error**
  + **Optional:**No.
  + **Nullable**: No
  + **Type:**DOMError
  + **Description:**[DOMError](http://dvcs.w3.org/hg/domcore/raw-file/default/Overview.html#error-types-table): of type TimeoutError if the timeout timer expired, SecurityError if not authorized to use the service, AbortError if the discovery process was canceled by the application. In addition service type or filter specific error types could be issued.

### Interface Service

The Service interface provides an API to bind to a specific service and monitor the availability of a bound service in an asynchronous manner. The process of binding to a service involves:  
  
1. mutual authentication between the service and the personal zone  
2. in case of cross zone personal interworking, mutual authentication between the zones  
3. agreement on data handling obligations as set out in the service's privacy policy  
4. verifying access privileges and checks the need for elevated privileges  
5. instantiate an implementation of the API that can be used by applications to request services from the requested API.  
  
Once the service object is instantiated, the service object will act as a proxy to the remote peer that will be able to invoke methods associated with API type under the window object of the remote peer. For example an application that successfully binds to a "http://picosec.org/api/tv" API will be able to invoke methods from the remote peer as described in the code example below.  
  
The Service interface inherits the DOM EventTarget interface. This means that events can be dispatched at instances of the Service interface or on instances of interfaces that inherit the Service interface. One example is the picosec generic Sensor API Sensor interface that inherits the Service interface and listens to sensor events.

##### WebIDL

        [NoInterfaceObject] interface Service : [EventTarget](http://dvcs.w3.org/hg/domcore/raw-file/tip/Overview.html#eventtarget) {  
  
  
  
                readonly attribute ServiceState state;  
  
                  
  
  
  
                readonly attribute DOMString api;  
  
                  
  
  
  
                readonly attribute DOMString id;  
  
                  
  
  
  
                readonly attribute DOMString displayName;  
  
                  
  
  
  
                readonly attribute DOMString serviceAddress;              
  
                  
  
  
  
                readonly attribute DOMString description;  
  
                  
  
  
  
                readonly attribute DOMString icon;  
  
                  
  
  
  
                void bindService([BindCallBack](http://picosec.org:8080/specs/apis/servicediscovery.html#::BindCallBack) bindCallBack, optional DOMString serviceId);  
  
                  
  
  
  
                void unbindService();  
  
        };

##### Code example

        succesBindCallBack(tvService) {  
                // Invoke a remote method.  
                tvService.display.setChannel(channel, success);  
                  
                // Register an event listner from event orginating from the remote peer.  
                tvService.display.addEventListener('channelchange', success);  
        }  

#### Attributes

**readonly ServiceState state**

Current service state.

This attribute is readonly.

**readonly DOMString api**

API is a global unique URI identifying the type of API provided by the service.

This attribute is readonly.

**readonly DOMString id**

Id is a globally unique id representing the binding to the service. The id can be used to resume the binding again to the service without invoking the findServices process again. The id can be stored persistently to be able to resume a binding to a service across power cycles.

This attribute is readonly.

**readonly DOMString displayName**

A human readable name of the service.

This attribute is readonly.

**readonly DOMString serviceAddress**

A human readable name of the device hosting the service.

This attribute is readonly.

**readonly DOMString description**

A URL referring to a detailed description of the service.

This attribute is readonly.

**readonly DOMString icon**

Icon is an URL referring to an icon that represents the service.

This attribute is readonly.

#### Methods

**bindService**

Binds to the service uniquely identified by the service identity.

##### Signature

void bindService([BindCallBack](http://picosec.org:8080/specs/apis/servicediscovery.html#::BindCallBack) bindCallBack, optional DOMString serviceId);

##### Parameters

* **bindCallBack**
  + **Optional:**No.
  + **Nullable**: No
  + **Type:**[BindCallBack](http://picosec.org:8080/specs/apis/servicediscovery.html#::BindCallBack)
  + **Description:**Asynchrounous callback to report the states of the bindService operation and the availability of the service.
* **serviceId**
  + **Optional:**Yes.
  + **Nullable**: No
  + **Type:**DOMString
  + **Description:**Unique id of the binding to the particular service. If no serviceId is provided as an in parameter, the id attribute in the Service interface will be used to bind the service.

**unbindService**

Releases all resources and connections allocated by the service object.

##### Signature

void unbindService();

### Interface BindCallBack

Bind success callback interface definition

##### WebIDL

         callback interface BindCallBack {  
  
  
  
                 void onBind([Service](http://picosec.org:8080/specs/apis/servicediscovery.html#::Service) service);  
  
                   
  
  
  
                 void onUnbind([Service](http://picosec.org:8080/specs/apis/servicediscovery.html#::Service) service);  
  
                   
  
  
  
                 void onServiceAvailable([Service](http://picosec.org:8080/specs/apis/servicediscovery.html#::Service) service);  
  
                   
  
  
  
                 void onServiceUnavailable([Service](http://picosec.org:8080/specs/apis/servicediscovery.html#::Service) service);  
  
                            
  
  
  
                 void onError(DOMError error);   
  
        };

#### Methods

**onBind**

Asynchronous success callback.

##### Signature

void onBind([Service](http://picosec.org:8080/specs/apis/servicediscovery.html#::Service) service);

##### Parameters

* **service**
  + **Optional:**No.
  + **Nullable**: No
  + **Type:**[Service](http://picosec.org:8080/specs/apis/servicediscovery.html#::Service)
  + **Description:**An input argument representing the service.

**onUnbind**

Asynchronous callback used when a service is unbound.

##### Signature

void onUnbind([Service](http://picosec.org:8080/specs/apis/servicediscovery.html#::Service) service);

##### Parameters

* **service**
  + **Optional:**No.
  + **Nullable**: No
  + **Type:**[Service](http://picosec.org:8080/specs/apis/servicediscovery.html#::Service)
  + **Description:**An input argument representing the service.

**onServiceAvailable**

Asynchronous callback indicating that the service is available again.

##### Signature

void onServiceAvailable([Service](http://picosec.org:8080/specs/apis/servicediscovery.html#::Service) service);

##### Parameters

* **service**
  + **Optional:**No.
  + **Nullable**: No
  + **Type:**[Service](http://picosec.org:8080/specs/apis/servicediscovery.html#::Service)
  + **Description:**An input argument representing the service.

**onServiceUnavailable**

Asynchronous callback indicating the service is temporarily unavailable.

##### Signature

void onServiceUnavailable([Service](http://picosec.org:8080/specs/apis/servicediscovery.html#::Service) service);

##### Parameters

* **service**
  + **Optional:**No.
  + **Nullable**: No
  + **Type:**[Service](http://picosec.org:8080/specs/apis/servicediscovery.html#::Service)
  + **Description:**An input argument representing the service.

**onError**

Asynchronous error callback.

##### Signature

void onError(DOMError error);

##### Parameters

* **error**
  + **Optional:**No.
  + **Nullable**: No
  + **Type:**DOMError
  + **Description:**DOMError (see above for definitions used by this interface).

### Dictionary Options

Options dictionary

##### WebIDL

        dictionary Options {  
  
  
  
                unsigned short timeout = 120;  
  
        };

#### Dictionary Members

**unsigned short timeout**

A timeout value for the findService operation in seconds between 0-65535. Default value is 120 seconds. It is possible to disable the timeout by setting the timeout value to Number.POSITIVE\_INFINITY.

### Dictionary Filter

Filter dictionary. An instance of this dictionary can optionally be used in findServices(). All dictionary member fields are optional except that the the zoneId dictionary member must be present when the cache member is set to "false".

##### WebIDL

        dictionary Filter {  
  
  
  
  
  
                boolean cache = true;     
  
          
  
  
  
                DOMString[] zoneId;  
  
                  
  
  
  
                DOMString serviceID;  
  
                   
  
  
  
                boolean remoteServices = false;  
  
  
  
  
  
                [ServiceLocation](http://picosec.org:8080/specs/apis/servicediscovery.html#::ServiceLocation)? serviceLocation;        
  
        };

##### Code example

       // Examples of picosec service discovery using the filter parameter.             
                                        
       // Initiate a search query for a temperature sensor in two personal zones.  
       findHandle = window.picosec.discovery.findServices(  
                    {api:'http://picosec.org/api/sensors/temperature'},   
                    {onFound:serviceFoundCB, onLost:serviceLostCB},  
                    null,  
                    {cache:false, zoneId:[bobsmith@pzh.picosec.org/raspberrypi, tobyealden@pzh.picosec.org/raspberrypi]});                                        
                                        
       // Initiate a search query for a specific TV service whose service ID is already known.  
       findHandle = window.picosec.discovery.findServices(  
                    {api:'http://picosec.org/api/tv'},   
                    {onFound:serviceFoundCB, onLost:serviceLostCB},  
                    null,  
                    {serviceID: myServiceId});                                                                                                                          
             

#### Dictionary Members

**boolean cache**

When cache is true, which is default, it means that a call to findServices () only will search for information on available services cached in the PZP on the users current device. This means that no communication with the PZH or with other devices are executed.  
When cache is false, it means that a call to findServices () will execute communication, either through the PZH or directly, with the actual external devices within the user's personal zone or in other personal zones to check the actual availability of the searched service. When cache is false the zoneId dictionary member must be present to limit the search to the specified personal zones.

**DOMString[] zoneId**

Identities of personal zones that will be used to search for services.   
If this dictionary member is not present the service will be searched for in all available zones.  
If the dictionary member is present but no one of the zones listed are available an onError callback, with DOMError type "NetworkError", is issued.

**DOMString serviceID**

If set service discovery will only return the service with the given identifier. This can be used to establish a service binding directly if the service identity is already known, for example if the service identity was stored persistently for later reuse or it was received via out of band.

**boolean remoteServices**

If remoteService is false the findServices method will limit the search for services that are connected directly to the device (including Bluetooth, NFC, etc if available) or to the same local IP network.  
If remoteServices is true, the findServices method will extend the search for services outside the local IP network (including other PZHs). Default value is false.

[**ServiceLocation**](http://picosec.org:8080/specs/apis/servicediscovery.html#::ServiceLocation)**? serviceLocation**

With the serviceLocation attribute it is possible to indicate where the service shall be located. If the serviceLocation dictionary member is not present or null, the location of the service is not considered during the findServices process.

### Dictionary ServiceLocation

ServiceLocation interface

##### WebIDL

        dictionary ServiceLocation {  
  
  
  
                double? latitude;  
  
  
  
  
  
                double? longitude;  
  
                  
  
  
  
                double accuracy;                  
  
        };

#### Dictionary Members

**double? latitude**

The latitude attribute is the geographic coordinate specified in decimal degrees. If the latitude is null the latitude of the device invoking the findServices method will be used.

**double? longitude**

The longitude attribute is the geographic coordinate specified in decimal degrees. If the longitude is null the longitude of the device invoking the findServices method will be used.

**double accuracy**

The accuracy denotes the accuracy level of the latitude and longitude coordinates in meters. This is used to limit the geographical area for finding services.

### Interface PendingOperation

Pending Operation interface

##### WebIDL

        [NoInterfaceObject] interface PendingOperation {  
  
  
  
                void cancel();  
  
        };

#### Methods

**cancel**

Method Cancel

##### Signature

void cancel();

Cancels the pending asynchronous discovery operation. When this method is called, the user agent must immediately bring the operation to a stop and return. Allocated resources should be released and any ongoing related network operations should be terminated. An error callback is issued with the DOMError name "AbortError".

### Interface picosec (partial interface)

The picosecDiscovery interface describes the part of the Discovery API accessible through the picosec object.

##### WebIDL

        partial interface picosec {  
  
                readonly attribute [DiscoveryInterface](http://picosec.org:8080/specs/apis/servicediscovery.html#::DiscoveryInterface) discovery;  
  
        };

## Enums

### ServiceState

Service state.

##### WebIDL

        enum ServiceState {"initiating", "available", "unavailable"};

#### Values

initiating

available

unavailable

## Features

This section lists the URIs used to declare the features of this API. The feature URIs are used by the developer in:

* The application's config.xml file to declare requested features.
* As identifier for serviceType in the picosec Discovery API's findServices() method.

**http://picosec.org/api/discovery**

## Full WebIDL

##### WebIDL

        [NoInterfaceObject] interface DiscoveryInterface  {  
  
  
  
                 [PendingOperation](http://picosec.org:8080/specs/apis/servicediscovery.html#::PendingOperation) findServices(    
  
                        [ServiceType](http://picosec.org:8080/specs/apis/servicediscovery.html#::ServiceType) serviceType,   
  
                        [FindCallBack](http://picosec.org:8080/specs/apis/servicediscovery.html#::FindCallBack) findCallBack,   
  
                        optional [Options](http://picosec.org:8080/specs/apis/servicediscovery.html#::Options)? options,   
  
                        optional [Filter](http://picosec.org:8080/specs/apis/servicediscovery.html#::Filter)? filter);  
  
                   
  
  
  
                 DOMString getServiceId(DOMString serviceType);  
  
                   
  
        };                
  
  
  
        dictionary ServiceType {  
  
  
  
                DOMString api;  
  
        };  
  
                  
  
  
  
        callback interface FindCallBack {  
  
  
  
                 void onFound([Service](http://picosec.org:8080/specs/apis/servicediscovery.html#::Service) service);          
  
                   
  
  
  
                 void onLost([Service](http://picosec.org:8080/specs/apis/servicediscovery.html#::Service) service);  
  
                            
  
  
  
                 void onError(DOMError error);  
  
                                   
  
        };  
  
          
  
  
  
        enum ServiceState {"initiating", "available", "unavailable"};  
  
          
  
  
  
          
  
        [NoInterfaceObject] interface Service : [EventTarget](http://dvcs.w3.org/hg/domcore/raw-file/tip/Overview.html#eventtarget) {  
  
  
  
                readonly attribute ServiceState state;  
  
                  
  
  
  
                readonly attribute DOMString api;  
  
                  
  
  
  
                readonly attribute DOMString id;  
  
                  
  
  
  
                readonly attribute DOMString displayName;  
  
                  
  
  
  
                readonly attribute DOMString serviceAddress;              
  
                  
  
  
  
                readonly attribute DOMString description;  
  
                  
  
  
  
                readonly attribute DOMString icon;  
  
                  
  
  
  
                void bindService([BindCallBack](http://picosec.org:8080/specs/apis/servicediscovery.html#::BindCallBack) bindCallBack, optional DOMString serviceId);  
  
                  
  
  
  
                void unbindService();  
  
        };  
  
          
  
  
  
           
  
         callback interface BindCallBack {  
  
  
  
                 void onBind([Service](http://picosec.org:8080/specs/apis/servicediscovery.html#::Service) service);  
  
                   
  
  
  
                 void onUnbind([Service](http://picosec.org:8080/specs/apis/servicediscovery.html#::Service) service);  
  
                   
  
  
  
                 void onServiceAvailable([Service](http://picosec.org:8080/specs/apis/servicediscovery.html#::Service) service);  
  
                   
  
  
  
                 void onServiceUnavailable([Service](http://picosec.org:8080/specs/apis/servicediscovery.html#::Service) service);  
  
                            
  
  
  
                 void onError(DOMError error);   
  
        };  
  
          
  
  
  
        dictionary Options {  
  
  
  
                unsigned short timeout = 120;  
  
        };  
  
          
  
  
  
        dictionary Filter {  
  
  
  
  
  
                boolean cache = true;     
  
          
  
  
  
                DOMString[] zoneId;  
  
                  
  
  
  
                DOMString serviceID;  
  
                   
  
  
  
                boolean remoteServices = false;  
  
  
  
  
  
                [ServiceLocation](http://picosec.org:8080/specs/apis/servicediscovery.html#::ServiceLocation)? serviceLocation;        
  
        };  
  
          
  
  
  
        dictionary ServiceLocation {  
  
  
  
                double? latitude;  
  
  
  
  
  
                double? longitude;  
  
                  
  
  
  
                double accuracy;                  
  
        };  
  
          
  
          
  
  
  
        [NoInterfaceObject] interface PendingOperation {  
  
  
  
                void cancel();  
  
        };  
  
          
  
  
  
  
  
        partial interface picosec {  
  
                readonly attribute [DiscoveryInterface](http://picosec.org:8080/specs/apis/servicediscovery.html#::DiscoveryInterface) discovery;  
  
        };

# Invocation

An IOT device must be able to emit sensor readings and and IOT application must be able to consume the same.

A messaging protocols such as MQTT, obeying a publish subscribe pattern is capable of implementing this minimal requirement.

However there are many instances where this basic method is insufficient.

For example

* Discovery: discovery is a request against one or more devices, which can be responded to either synchronously or asynchronously.
* Administration: device configuration request are levied against a device and need to be responded to in clear fashion
* Database requests: iot-devices are not necessarily connected permanently to the internet. It is a requirement to be able to both cache data on device and respond to data queries
* Complex data requests: it is a frequent requirement to be able to request data “pushes” only when certain criteria are fulfilled. This is simply to optimse power and networking resource hogging. For example
  + emit location ping when exiting an geofence
  + throttle data updates so that only interesting data points are emitted
  + emit only deltas exceeding a predetermined amount.

For these and other reasons it is useful to have a general purpose invocation mechanism where an IOT application can make are remote API request against a device and vice versa.

This mechanism must support asynchronous responses.

We currently have two mechanisms in scope to handle this requirement.

## JSON-RPC

JSON-RPC is a simple API remoting mechanism designed to be lightweight and simple.

### Overview

JSON-RPC is a stateless, light-weight remote procedure call (RPC) protocol. Primarily this specification defines several data structures and the rules around their processing. It is transport agnostic in that the concepts can be used within the same process, over sockets, over http, or in many various message passing environments. It uses JSON (RFC 4627) as data format.

It is designed to be simple!

### Conventions

The key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY”, and “OPTIONAL” in this document are to be interpreted as described in RFC 2119.

Since JSON-RPC utilizes JSON, it has the same type system (see [http://www.json.org](http://www.json.org/) or RFC 4627). JSON can represent four primitive types (Strings, Numbers, Booleans, and Null) and two structured types (Objects and Arrays). The term “Primitive” in this specification references any of those four primitive JSON types. The term “Structured” references either of the structured JSON types. Whenever this document refers to any JSON type, the first letter is always capitalized: Object, Array, String, Number, Boolean, Null. True and False are also capitalized.

All member names exchanged between the Client and the Server that are considered for matching of any kind should be considered to be case-sensitive. The terms function, method, and procedure can be assumed to be interchangeable.

The Client is defined as the origin of Request objects and the handler of Response objects. The Server is defined as the origin of Response objects and the handler of Request objects.

One implementation of this specification could easily fill both of those roles, even at the same time, to other different clients or the same client. This specification does not address that layer of complexity.

### Compatibility

JSON-RPC 2.0 Request objects and Response objects may not work with existing JSON-RPC 1.0 clients or servers. However, it is easy to distinguish between the two versions as 2.0 always has a member named “jsonrpc” with a String value of “2.0” whereas 1.0 does not. Most 2.0 implementations should consider trying to handle 1.0 objects, even if not the peer-to-peer and class hinting aspects of 1.0.

### Request object

A rpc call is represented by sending a Request object to a Server. The Request object has the following members:

**jsonrpc** A String specifying the version of the JSON-RPC protocol. MUST be exactly “2.0”. **method** A String containing the name of the method to be invoked. Method names that begin with the word rpc followed by a period character (U+002E or ASCII 46) are reserved for rpc-internal methods and extensions and MUST NOT be used for anything else. **params** A Structured value that holds the parameter values to be used during the invocation of the method. This member MAY be omitted. **id** An identifier established by the Client that MUST contain a String, Number, or NULL value if included. If it is not included it is assumed to be a notification. The value SHOULD normally not be Null [1] and Numbers SHOULD NOT contain fractional parts [2] The Server MUST reply with the same value in the Response object if included. This member is used to correlate the context between the two objects.

[1] The use of Null as a value for the id member in a Request object is discouraged, because this specification uses a value of Null for Responses with an unknown id. Also, because JSON-RPC 1.0 uses an id value of Null for Notifications this could cause confusion in handling.

[2] Fractional parts may be problematic, since many decimal fractions cannot be represented exactly as binary fractions.

#### Notification

A Notification is a Request object without an “id” member. A Request object that is a Notification signifies the Client’s lack of interest in the corresponding Response object, and as such no Response object needs to be returned to the client. The Server MUST NOT reply to a Notification, including those that are within a batch request.

Notifications are not confirmable by definition, since they do not have a Response object to be returned. As such, the Client would not be aware of any errors (like e.g. “Invalid params”,”Internal error”).

#### Parameter Structures

If present, parameters for the rpc call MUST be provided as a Structured value. Either by-position through an Array or by-name through an Object.

* by-position: params MUST be an Array, containing the values in the Server expected order.
* by-name: params MUST be an Object, with member names that match the Server expected parameter names. The absence of expected names MAY result in an error being generated. The names MUST match exactly, including case, to the method’s expected parameters.

### Response object

When a rpc call is made, the Server MUST reply with a Response, except for in the case of Notifications. The Response is expressed as a single JSON Object, with the following members:

**jsonrpc** A String specifying the version of the JSON-RPC protocol. MUST be exactly “2.0”. **result** This member is REQUIRED on success. This member MUST NOT exist if there was an error invoking the method. The value of this member is determined by the method invoked on the Server. **error** This member is REQUIRED on error. This member MUST NOT exist if there was no error triggered during invocation. The value for this member MUST be an Object as defined in section 5.1. **id** This member is REQUIRED. It MUST be the same as the value of the id member in the Request Object. If there was an error in detecting the id in the Request object (e.g. Parse error/Invalid Request), it MUST be Null.

Either the result member or error member MUST be included, but both members MUST NOT be included.

#### Error object

When a rpc call encounters an error, the Response Object MUST contain the error member with a value that is a Object with the following members:

**code** A Number that indicates the error type that occurred. This MUST be an integer. **message** A String providing a short description of the error. The message SHOULD be limited to a concise single sentence. **data** A Primitive or Structured value that contains additional information about the error. This may be omitted.

The value of this member is defined by the Server (e.g. detailed error information, nested errors etc.).

The error codes from and including -32768 to -32000 are reserved for pre-defined errors. Any code within this range, but not defined explicitly below is reserved for future use. The error codes are nearly the same as those suggested for XML-RPC at the following url: <http://xmlrpc-epi.sourceforge.net/specs/rfc.fault_codes.php>

code message meaning -32700 Parse error Invalid JSON was received by the server. An error occurred on the server while parsing the JSON text. -32600 Invalid Request The JSON sent is not a valid Request object. -32601 Method not found The method does not exist / is not available. -32602 Invalid params Invalid method parameter(s). -32603 Internal error Internal JSON-RPC error. -32000 to -32099 Server error Reserved for implementation-defined server-errors. The remainder of the space is available for application defined errors.

### Batch

To send several Request objects at the same time, the Client MAY send an Array filled with Request objects.

The Server should respond with an Array containing the corresponding Response objects, after all of the batch Request objects have been processed. A Response object SHOULD exist for each Request object, except that there SHOULD NOT be any Response objects for notifications. The Server MAY process a batch rpc call as a set of concurrent tasks, processing them in any order and with any width of parallelism.

The Response objects being returned from a batch call MAY be returned in any order within the Array. The Client SHOULD match contexts between the set of Request objects and the resulting set of Response objects based on the id member within each Object.

If the batch rpc call itself fails to be recognized as an valid JSON or as an Array with at least one value, the response from the Server MUST be a single Response object. If there are no Response objects contained within the Response array as it is to be sent to the client, the server MUST NOT return an empty Array and should return nothing at all.

### Examples

Syntax:

--> data sent to Server

<-- data sent to Client

rpc call with positional parameters:

--> {"jsonrpc": "2.0", "method": "subtract", "params": [42, 23], "id": 1}

<-- {"jsonrpc": "2.0", "result": 19, "id": 1}

--> {"jsonrpc": "2.0", "method": "subtract", "params": [23, 42], "id": 2}

<-- {"jsonrpc": "2.0", "result": -19, "id": 2}

rpc call with named parameters:

--> {"jsonrpc": "2.0", "method": "subtract", "params": {"subtrahend": 23, "minuend": 42}, "id": 3}

<-- {"jsonrpc": "2.0", "result": 19, "id": 3}

--> {"jsonrpc": "2.0", "method": "subtract", "params": {"minuend": 42, "subtrahend": 23}, "id": 4}

<-- {"jsonrpc": "2.0", "result": 19, "id": 4}

a Notification:

--> {"jsonrpc": "2.0", "method": "update", "params": [1,2,3,4,5]}

--> {"jsonrpc": "2.0", "method": "foobar"}

rpc call of non-existent method:

--> {"jsonrpc": "2.0", "method": "foobar", "id": "1"}

<-- {"jsonrpc": "2.0", "error": {"code": -32601, "message": "Method not found"}, "id": "1"}

rpc call with invalid JSON:

--> {"jsonrpc": "2.0", "method": "foobar, "params": "bar", "baz]

<-- {"jsonrpc": "2.0", "error": {"code": -32700, "message": "Parse error"}, "id": null}

rpc call with invalid Request object:

--> {"jsonrpc": "2.0", "method": 1, "params": "bar"}

<-- {"jsonrpc": "2.0", "error": {"code": -32600, "message": "Invalid Request"}, "id": null}

rpc call Batch, invalid JSON:

--> [

{"jsonrpc": "2.0", "method": "sum", "params": [1,2,4], "id": "1"},

{"jsonrpc": "2.0", "method"

]

<-- {"jsonrpc": "2.0", "error": {"code": -32700, "message": "Parse error"}, "id": null}

rpc call with an empty Array:

--> []

<-- {"jsonrpc": "2.0", "error": {"code": -32600, "message": "Invalid Request"}, "id": null}

rpc call with an invalid Batch (but not empty):

--> [1]

<-- [

{"jsonrpc": "2.0", "error": {"code": -32600, "message": "Invalid Request"}, "id": null}

]

rpc call with invalid Batch:

--> [1,2,3]

<-- [

{"jsonrpc": "2.0", "error": {"code": -32600, "message": "Invalid Request"}, "id": null},

{"jsonrpc": "2.0", "error": {"code": -32600, "message": "Invalid Request"}, "id": null},

{"jsonrpc": "2.0", "error": {"code": -32600, "message": "Invalid Request"}, "id": null}

]

rpc call Batch:

--> [

{"jsonrpc": "2.0", "method": "sum", "params": [1,2,4], "id": "1"},

{"jsonrpc": "2.0", "method": "notify\_hello", "params": [7]},

{"jsonrpc": "2.0", "method": "subtract", "params": [42,23], "id": "2"},

{"foo": "boo"},

{"jsonrpc": "2.0", "method": "foo.get", "params": {"name": "myself"}, "id": "5"},

{"jsonrpc": "2.0", "method": "get\_data", "id": "9"}

]

<-- [

{"jsonrpc": "2.0", "result": 7, "id": "1"},

{"jsonrpc": "2.0", "result": 19, "id": "2"},

{"jsonrpc": "2.0", "error": {"code": -32600, "message": "Invalid Request"}, "id": null},

{"jsonrpc": "2.0", "error": {"code": -32601, "message": "Method not found"}, "id": "5"},

{"jsonrpc": "2.0", "result": ["hello", 5], "id": "9"}

]

rpc call Batch (all notifications):

--> [

{"jsonrpc": "2.0", "method": "notify\_sum", "params": [1,2,4]},

{"jsonrpc": "2.0", "method": "notify\_hello", "params": [7]}

]

<-- //Nothing is returned for all notification batches

### Extensions

Method names that begin with rpc. are reserved for system extensions, and MUST NOT be used for anything else. Each system extension is defined in a related specification. All system extensions are OPTIONAL.

### JSON-RPC bindings

#### Routing Algorithm

##### Message validation

The message metadata shall be checked at both sender/publisher and receiver/subscriber sides to ensure validity, well-formedness and consistency.

Message validation failure MAY result in automatic generation and sending of delivery notification messages by the checking party if it’s not the message source and the message requires delivery notification. This kind of behaviour, however, is implementation-defined. Failure to validate a message SHALL prevent the message from being further processed.

Errors during the validation phase when the message source is local SHALL, instead, result in appropriate exceptions being thrown at the application-level.

##### Route Determination

The route determination decides how each recipient is to be reached. In the route determination phase the routing manager SHALL check the recipient and determine whether it is local or remote. In the former case, it SHALL ensure that the recipient actually exists and is reachable upon policy restrictions and, if this condition does not hold true, it SHOULD automatically generate and send a delivery notification message to inform the message source. In the latter case it SHALL determine whether the recipient is reachable within the local networks the device is connected to or not, and also which network interface is going to be used to reach the recipient, so that this information is available in the policy enforcement phase.

Route determination for messages SHALL be performed once per recipient, so that individual failures SHALL only affect one message/recipient combination. Failures during the route determination phase SHALL prevent the message from being further processed for that message/recipient combination.

##### Application-level routing

The message handling functionality exposed to the application, at the lowest level, consists of a restricted set of conceptually simple operations: generating and sending messages, or forwarding them, and registering listeners for incoming messages.

This implies that the message routing manager SHALL be able to somehow create the JavaScript representations of incoming message objects and to properly trigger the execution of the relevant listener callbacks.

Each application SHALL be allowed to handle messages from/to addressable entities that it “owns” (e.g., the application itself, the services it exposes), obviously according to the constraints imposed by local policy rules and authorizations, hence it SHALL be able to both receive messages directed to any addressable entity that it owns and to send/forward message on the behalf of any addressable entity it owns.

Other than that, at this level, the behaviour of the system is highly dependent on the actual webinos inter-application messaging API specification, hence that document is to be considered the reference for determining whether a given implementation is in conformance or not.

##### Network-level routing

The network-level routing algorithm SHALL be based on the defined addressing scheme.

Each routing entity SHALL register itself with its parent node and child node if the other parties have not registered themselves with this routing entity. A successful registration process SHALL result in an entry or record in local routing table for both parties that registering and being registered with.

On receiving a message, the receiver checks if itself is the message destination. If so, based on the message type, correspondent handler functions/callbacks MAY be called. For example, if a JSON RPC type of message is received, the following procedure May happen:

* Message destination calls rpchandler to handle the message.
* RPC result is packed within a new message object. Return route back to message origin follows the same routing rule as any message experienced.
* When RPC result is received by the message origin, if a callback was defined during creating message, this callback will be executed.

In the case that the message receiver is not the final destination, the receiver SHALL forward the message. When there is no routing information about the final destination available at this message receiver, it SHALL look up the routing table for the next available hop to the destination.

#### Message Object

##### Generic Message Object

Each message results in the creation of a message object. A generic message object is defined here. Any user or system specified message object will be a subclass of the generic message object. The generic message object includes the following attributes:

| **Attribute** | **Description** | **Mandatory** | **Data type** |
| --- | --- | --- | --- |
| type | Identifies the message type and determines payload semantics | Yes | String that matches the following regular expression: [\_a-zA-Z][\_a-zA-Z0-9]\* |
| from | Represents the entity that originally sent the message | Yes | Addressable entity object reference |
| to | destination entity | Yes | addressable entity object reference |
| id | Message identifier | Yes | String identifying the message |
| timestamp | Moment in time in which the message is generated by the original message source | No | A suitable date/time representation with millisecond precision or better |
| expires | Moment in time past which the message is no more valid or meaningful | No | A suitable date/time representation with millisecond precision or better |
| deliveryReceipt | Indicates whether the sending entity wants to be notified by the receiving entities about the result of the message delivery | No | Boolean value |
| payload | Message type-specific data | No | String |

This specification does not demand the usage of a specific serialization for transmitting messages, yet it presupposes that:

* the actual serialization has a one-to-one mapping with Unicode code points for addresses and payload data;
* a one-to-one mapping between addressable entities and address strings is defined.

The ‘id’ attribute is used by the sender to track if a response or error message that it might receive is in relation to that message.

It is up to the sender of the message whether the value of the ‘id’ attribute is unique only within the current session or is globally unique.

#### Delivery Notification Message object

The delivery notification protocol is part of the core webinos message handling protocol and it provides a handful of ICMP-like functionality to the system.

A delivery notification message SHOULD be generated when an entity receives or is about to receive an message having the “deliveryReceipt” attribute specified as true. The notification message MAY be generated at the destination of message delivery or at the router on the path when there is issue to pass the message on, e.g. message validation fails or the router couldn’t find next hop to deliver the message.

The Delivery notification message object is a subclass of generic message object. In order for a delivery notification message to be considered valid, the following attributes MUST be set as hereby described:

| **Attribute** | **Valid values** |
| --- | --- |
| type | “deliveryNotification” |
| to | The “to” MUST be the entity that this notification was generated or, if none, the entity is specified by the Source attribute |
| id | Reference to the message because of which this notification was generated |
| deliveryReceipt | false |
| payload | String |

The Payload attribute MUST be specified as one of the following values:

| **Value** | **Meaning** |
| --- | --- |
| “ok” | Message successfully delivered |
| “duplicate” | A message with the same ID was already delivered to the recipient |
| “invalid” | The recipient got an invalid message (e.g., transmission error) |
| “badDestination” | The intended recipient is unknown or unreachable |
| “expired” | The message expired before the actual delivery, according to its “Expiry timestamp” attribute |
| “refused” | The message could not be received because of lack of authorization and/or policy settings |
| “noReference” | The recipient does not hold a local reference to the message specified by the “In response to” attribute |

#### RPC Message Object

The RPC Message Object message object is also a subclass of generic message object. It basically wraps [[Messaging\_and\_Routing#51-JSONRPC2.0|JSONRPC2.0]] objects and batches into webinos messages, hence concepts, terminology, payload syntax and semantics, and behaviour are meant to be aligned with that specification.

A JSON-RPC 2.0 request object or request batch is wrapped into a webinos RPC request message that, in order to be considered valid, MUST have the following attributes set as hereby described:

| **Attribute** | **Valid values** |
| --- | --- |
| type | “JSONRPC20Request” |
| id | Identity of the RPC request message |
| payload | JSON-RPC 2.0 request object or request batch |

When an entity offering RPC interfaces receives an RPC request message that does not contain a notification or a notification-only batch and said entity is a primary recipient of that message, it SHOULD generate and send an RPC response message, possibly according to specific policy rules.

Similarly to webinos RPC request messages, a JSON-RPC 2.0 response object or response batch is wrapped into a webinos RPC response message that, in order to be considered valid, MUST have the following attributes set as hereby described:

| **Attribute** | **Valid values** |
| --- | --- |
| type | “JSONRPC20Response” |
| to | The “to” set MUST contain only a reference to the entity specified by the “to” attribute in the RPC request message because of which this RPC response message was generated |
| id | Reference to the RPC request message because of which this RPC response message was generated |
| payload | JSON-RPC 2.0 response object or response batch |

Note: the JSON-RPC 2.0 specification states that notifications MUST NOT be replied to, and the same holds true for this specification; nevertheless it is still possible to use the “deliveryReceipt” attribute to be informed about the delivery of an RPC request message, even if it contains a notification or a notification-only batch.

This protocol is meant to be automatically handled by the WRT without requiring any extra effort on the developer side: the message Handling API and the routing manager SHALL prevent applications from creating and receiving messages whose type is “JSONRPC20Request” or “JSONRPC20Response”, while it SHALL be possible to expose RPC interfaces by describing them in the config.xml file, as specified in the Foundations section of this specification, and the WRT SHALL automatically create functions with corresponding names into the local JavaScript object that represents a discovered service, mapping calls to those functions to asynchronous RPC requests and responses operated via this protocol.

#### PROP Message Object

PROP message type is created by webinos for TLS session set-up and communications between webinos core components. It MUST have the following attributes set as hereby described:

| **Attribute** | **Valid values** |
| --- | --- |
| type | “Prop” |
| to | destination entity |
| id | Reference to the Prop message |
| payload | Prop message body |

<http://www.jsonrpc.org/specification>

## Optional BSON binding

<http://bsonspec.org/>

BSON [bee · sahn], short for Bin­ary JSON, is a bin­ary-en­coded seri­al­iz­a­tion of JSON-like doc­u­ments. Like JSON, BSON sup­ports the em­bed­ding of doc­u­ments and ar­rays with­in oth­er doc­u­ments and ar­rays. BSON also con­tains ex­ten­sions that al­low rep­res­ent­a­tion of data types that are not part of the JSON spec. For ex­ample, BSON has a Date type and a BinData type.

BSON can be com­pared to bin­ary inter­change for­mats, like Proto­col Buf­fers. BSON is more “schema-less” than Proto­col Buf­fers, which can give it an ad­vant­age in flex­ib­il­ity but also a slight dis­ad­vant­age in space ef­fi­ciency (BSON has over­head for field names with­in the seri­al­ized data).

BSON may be required in IOT devices for efficiency purposes

The full specification

<http://bsonspec.org/spec.html>

#### Specification Version 1.0

BSON is a binary format in which zero or more key/value pairs are stored as a single entity. We call this entity a document.

The following grammar specifies version 1.0 of the BSON standard. We’ve written the grammar using a pseudo-BNF syntax. Valid BSON data is represented by the document non-terminal.

#### Basic Types

The following basic types are used as terminals in the rest of the grammar. Each type must be serialized in little-endian format.

| **byte** | **1 byte (8-bits)** |
| --- | --- |
| int32 | 4 bytes (32-bit signed integer, two’s complement) |
| int64 | 8 bytes (64-bit signed integer, two’s complement) |
| double | 8 bytes (64-bit IEEE 754 floating point) |

#### Non-terminals

The following specifies the rest of the BSON grammar. Note that quoted strings represent terminals, and should be interpreted with C semantics (e.g.\x01 represents the byte 0000 0001). Also note that we use the operator as shorthand for repetition (e.g. (*\x01*2) is \x01\x01). When used as a unary operator, \* means that the repetition can occur 0 or more times.

document ::= int32 e\_list "\x00" BSON Document. int32 is the total number of bytes comprising the document.

e\_list ::= element e\_list

| ""

element ::= "\x01" e\_name double Floating point

| "\x02" e\_name string UTF-8 string

| "\x03" e\_name document Embedded document

| "\x04" e\_name document Array

| "\x05" e\_name binary Binary data

| "\x06" e\_name Undefined — Deprecated

| "\x07" e\_name (byte\*12) ObjectId

| "\x08" e\_name "\x00" Boolean "false"

| "\x08" e\_name "\x01" Boolean "true"

| "\x09" e\_name int64 UTC datetime

| "\x0A" e\_name Null value

| "\x0B" e\_name cstring cstring Regular expression - The first cstring is the regex pattern, the second is the regex options string. Options are identified by characters, which must be stored in alphabetical order. Valid options are 'i' for case insensitive matching, 'm' for multiline matching, 'x' for verbose mode, 'l' to make \w, \W, etc. locale dependent, 's' for dotall mode ('.' matches everything), and 'u' to make \w, \W, etc. match unicode.

| "\x0C" e\_name string (byte\*12) DBPointer — Deprecated

| "\x0D" e\_name string JavaScript code

| "\x0E" e\_name string Deprecated

| "\x0F" e\_name code\_w\_s JavaScript code w/ scope

| "\x10" e\_name int32 32-bit Integer

| "\x11" e\_name int64 Timestamp

| "\x12" e\_name int64 64-bit integer

| "\xFF" e\_name Min key

| "\x7F" e\_name Max key

e\_name ::= cstring Key name

string ::= int32 (byte\*) "\x00" String - The int32 is the number bytes in the (byte\*) + 1 (for the trailing '\x00'). The (byte\*) is zero or more UTF-8 encoded characters.

cstring ::= (byte\*) "\x00" Zero or more modified UTF-8 encoded characters followed by '\x00'. The (byte\*) MUST NOT contain '\x00', hence it is not full UTF-8.

binary ::= int32 subtype (byte\*) Binary - The int32 is the number of bytes in the (byte\*).

subtype ::= "\x00" Generic binary subtype

| "\x01" Function

| "\x02" Binary (Old)

| "\x03" UUID (Old)

| "\x04" UUID

| "\x05" MD5

| "\x80" User defined

code\_w\_s ::= int32 string document Code w/ scope

Notes

* Array - The document for an array is a normal BSON document with integer values for the keys, starting with 0 and continuing sequentially. For example, the array [‘red’, ‘blue’] would be encoded as the document {‘0’: ‘red’, ‘1’: ‘blue’}. The keys must be in ascending numerical order.
* UTC datetime - The int64 is UTC milliseconds since the Unix epoch. Timestamp - Special internal type used by MongoDB replication and sharding. First 4 bytes are an increment, second 4 are a timestamp.
* Min key - Special type which compares lower than all other possible BSON element values.
* Max key - Special type which compares higher than all other possible BSON element values.
* Generic binary subtype - This is the most commonly used binary subtype and should be the ‘default’ for drivers and tools.
* The BSON “binary” or “BinData” datatype is used to represent arrays of bytes. It is somewhat analogous to the Java notion of a ByteArray. BSON binary values have a subtype. This is used to indicate what kind of data is in the byte array. Subtypes from zero to 127 are predefined or reserved. Subtypes from 128-255 are user-defined.
* \x02 Binary (Old) - This used to be the default subtype, but was deprecated in favor of \x00. Drivers and tools should be sure to handle \x02 appropriately. The structure of the binary data (the bytearray in the binary non-terminal) must be an int32 followed by a (byte). The int32 is the number of bytes in the repetition.
* \x03 UUID (Old) - This used to be the UUID subtype, but was deprecated in favor of \x04. Drivers and tools for languages with a native UUID type should handle \x03 appropriately.
* \x80-0xff “User defined” subtypes. The binary data can be anything.
* Code w/ scope - The int32 is the length in bytes of the entire code\_w\_s value. The string is JavaScript code. The document is a mapping from identifiers to values, representing the scope in which the string should be evaluated.

## CoAP

The CoAP full specification is found on the IETF website <https://tools.ietf.org/html/rfc7252>

CoAP is based on HTTP, and inherits the RESTful model <http://en.wikipedia.org/wiki/Representational_state_transfer> for remote API invocation, but without many of the overheads implied by HTTP due to its efficient implementation on top of UDP

The aspects of the CoAP specification are repeated belwo

### Request/Response Semantics

CoAP operates under a similar request/response model as HTTP: a CoAP endpoint in the role of a “client” sends one or more CoAP requests to a “server”, which services the requests by sending CoAP responses. Unlike HTTP, requests and responses are not sent over a previously established connection but are exchanged asynchronously over CoAP messages.

### Requests

A CoAP request consists of the method to be applied to the resource, the identifier of the resource, a payload and Internet media type (if any), and optional metadata about the request.

CoAP supports the basic methods of GET, POST, PUT, and DELETE, which are easily mapped to HTTP. They have the same properties of safe (only retrieval) and idempotent (you can invoke it multiple times with the same effects) as HTTP (see Section 9.1 of [RFC2616]). The GET method is safe; therefore, it MUST NOT take any other action on a resource other than retrieval. The GET, PUT, and DELETE methods MUST be performed in such a way that they are idempotent. POST is not idempotent, because its effect is determined by the origin server and dependent on the target resource; it usually results in a new resource being created or the target resource being updated.

A request is initiated by setting the Code field in the CoAP header of a Confirmable or a Non-confirmable message to a Method Code and including request information.

The methods used in requests are described in detail in Section 5.8.

### Responses

After receiving and interpreting a request, a server responds with a CoAP response that is matched to the request by means of a client- generated token (Section 5.3); note that this is different from the Message ID that matches a Confirmable message to its Acknowledgement.

A response is identified by the Code field in the CoAP header being set to a Response Code. Similar to the HTTP Status Code, the CoAP Response Code indicates the result of the attempt to understand and satisfy the request. These codes are fully defined in Section 5.9. The Response Code numbers to be set in the Code field of the CoAP header are maintained in the CoAP Response Code Registry (Section 12.1.2).

### API bindings

A number of initiatives have already started looking at how CoAP binds generally to RESTful services

<https://datatracker.ietf.org/doc/rfc6690/?include_text=1>

<https://datatracker.ietf.org/wg/core/documents/>

## Work to do

CoAP webinos alignment

# Device Admin requirements

An IOT device needs updating. In the first instance we will restrict the scope of IOT administration to configuration only

See below for firmware update considerations

## iot-device administration

A iot-device is not required to support webinos synchronization. Therefore any iot-device configuration operations that are performed as standard using sync – particularly policy configuration and credentials configuration – must be accomplished by alternative means.

iot-device configuration requirements cover:

* common configuration requirements, notably access control policy;
* application-specific configuration requirements, including the configuration needs of the supported APIs and the connection technology.

A minimal RPC-based API is defined for remote administration of iot-devices. Support for this protocol is conditionally required: if a iot-device supports any remote configuration mechanism, it must at least support the iot-device Admin API. Certain iot-devices might be so limited as to be unable to support remote configuration, or their functionality might be so limited as to make it unnecessary.

Also, more powerful iot-devices might provide a locally-hosted web app for configuration; in this case, the authentication and access control requirements for such an interface are specified.

The API for common configuration requirements is specified here, and a generic settings mechanism is defined that may be used for application-specific configuration requirements. For particular applications, iot-devices may provide additional APIs for their specific configuration requirements.

## iot-device configuration API

Whilst it would be appealing for a iot-device to expose a REST configuration API, it would not be supportable, or a justifiable use of the available resources on the smaller iot-devices. Instead, a very simple JSON-RPC-based configuration API is defined, together with a mapping to a REST API that can be implemented at a proxy if desired.

The configuration API is defined to be as simple as possible. There is a single flat space of configuration key strings. The operations supported by the API are:

* keys with primitive values:
  + support for string and number value types;
  + get and set operations;
* keys whose values are an unordered set of primitive values:
  + support for string and number member value types;
  + get and set operations on the collection as a whole;
  + add and remove operations on members of the collection.

The JSON-RPC interface is served as a webinos API in the same way as other webinos APIs. The API is not discoverable, but is accessed with well-known identifiers as follows:

* URI: @<http://webinos.org/api/pzp-admin@>
* Instance: @default@

The API exposes a single interface Config, with the following operations.

### Config.get

Gets the primitive value of a configuration entry. An error is returned from this operation if one of the following conditions arises:

* the key is not known or not configurable on this iot-device;
* the caller does not have permission to read this key.

**Params**:

* key: string key of the configuration element;

**Result**:

* the JSON-encoded value of the configuration element. If the element has a primitive value, this is the JSON representation of the Number or String value. If the configuration element has a collection value, this is the JSON representation of an array containing the members of the collection; order is insignificant.

### JSON-RPC request

JSON: Object

{ id : 1

, jsonrpc : "2.0"

, method : "http://webinos.org/api/psp-admin@default.Config.get"

, params :

{ key :

}

}

### JSON-RPC response

JSON: Object

{ id: 1

, jsonrpc: "2.0"

, result:

}

### REST request

GET /config/

Accept: application/json

must be URL-encoded before forming the request path. A successful request results in a 200 response with a response body containing the JSON representation of the primitive or collection value. Error responses are indicated by a relevant status code and optionally an informative error message in the response body. ### Config.set Sets the primitive value of a configuration entry. An error is returned from this operation if one of the following conditions arises: \* the key is not known or not configurable on this iot-device; \* the caller does not have permission to modify this key; \* an invalid value or an value with invalid type is specified. \*\*Params\*\*: \* key: string key of the configuration element; \* value: the JSON-encoded value to set. If the configuration element has a primitive value, this is the JSON representation of the Number or String value. If the configuration element has a collection value, this is the JSON representation of an array containing the members of the collection; order is insignificant, and duplicate members are silently discarded. \*\*Result\*\*: \* none. #### JSON-RPC request

JSON: Object

{ id : 1

, jsonrpc : "2.0"

, method : "http://webinos.org/api/psp-admin@default.Config.set"

, params :

{ key :

, value :

}

}

#### REST request

PUT /config/

Content-Type: application/json

**key** must be URL-encoded before forming the request path. **value** is the JSON representation of the primitive or collection value.

A successful request results in a 200 response with a response body containing the JSON representation of the set primitive or collection value.

Error responses are indicated by a relevant status code and optionally an informative error message in the response body.

### Config.add

Inserts a new primitive value into a configuration entry of collection type.

An error is returned from this operation if one of the following conditions arises:

* the key is not known or not configurable on this iot-device;
* the caller does not have permission to modify this key;
* the key is not of collection type;
* an invalid value or an value with invalid type is specified.

If the given value is already present, the operation silently does nothing.

**Params**:

* key: string key of the configuration element;
* value: the JSON-encoded value to add. This is the JSON representation of the Number or String value.

**Result**:

* none.

#### JSON-RPC request

JSON: Object

{ id : 1

, jsonrpc : "2.0"

, method : "http://webinos.org/api/psp-admin@default.Config.add"

, params :

{ key :

, value :

}

}

#### REST request

POST /config/

Content-Type: application/json

**key** must be URL-encoded before forming the request path. **value** is the JSON representation of the primitive value.

A successful request results in a 201 response with a response body containing the JSON representation of the added primitive value.

Error responses are indicated by a relevant status code and optionally an informative error message in the response body.

### Config.remove

Removes a primitive value from a configuration entry of collection type.

An error is returned from this operation if one of the following conditions arises:

* the key is not known or not configurable on this iot-device;
* the caller does not have permission to modify this key;
* the key is not of collection type;
* an invalid value or an value with invalid type is specified.

If the given value is not a member of the collection, the operation silently does nothing.

**Params**:

* key: string key of the configuration element;
* value: the JSON-encoded value to remove. This is the JSON representation of the Number or String value.

**Result**:

* none.

#### JSON-RPC request

JSON: Object

{ id : 1

, jsonrpc : "2.0"

, method : "http://webinos.org/api/psp-admin@default.Config.add"

, params :

{ key :

, value :

}

}

#### REST request

DELETE /config//

Content-Type: application/json

<**key**> and <**value**> must each be URL-encoded before forming the request path.

A successful request results in a 200 response with a response body containing the JSON representation of the removed primitive value.

Error responses are indicated by a relevant status code and optionally an informative error message in the response body.

## TLS-capable iot-device

TLS-capable iot-device must expose the JSON-RPC admin API if it offers any remote configuration capability; any remotely configurable items must be configurable through that API.

A TLS-capable iot-device may also serve the REST API from a local HTTP server.

More capable iot-devices will typically serve an administration web app in addition to the API, allowing users for example to upload certificates directly from the admin web page.

## Tethered iot-device

A tethered iot-device must expose the JSON-RPC admin API if it offers any remote configuration capability; any remotely configurable items must be configurable through that API.

A proxy to a tethered iot-device may also expose the REST API, proxying requests to the JSON-RPC API.

## Admin API access control

The Admin API for a iot-device is clearly sensitive and must only be exposed to suitably authenticated clients. Once a device is enrolled, access to the admin API must be permitted only to clients in the same Personal Zone.

Prior to enrolment, however, a iot-device will typically initially be in a bootstrap mode and will have an unprotected admin interface, with that interface being used to set up and enrol the device, thereby establishing admin API access control, as part of commissioning the unit. A hardware reset would be required to regain access, also clearing all sensitive data, if relevant credentials were lost or compromised.

Devices that are not remotely configurable must have some means of locally connecting and configuring the device (eg a physical serial port dedicated to that purpose). No APIs are served in bootstrap mode. Local methods for enrolment depend on the nature of the device and its deployment, and are outside the scope of this specification.

In bootstrap mode, for devices supporting the JSON-RPC admin API, no APIs are available through the RPC port other than the admin API. The requirements for admin API access control, both in bootstrap and active mode, are different according to the class of iot-device.

### TLS-capable iot-device

In bootstrap mode, as well as in active mode, connections to the JSON\_RPC port are made over TLS. In bootstrap mode, the iot-device exposes a bootstrap server certificate issued by the manufacturer. The iot-device may require clients to authenticate using a client certificate when in bootstrap mode; if so, that client certificate and key must also be issued by the manufacturer. (This is analogous to devices being shipped with a temporary manufacturer-set admin password.)

Once the enrolment process has been completed using the admin API, the device transitions to active mode, and the bootstrap credentials, if any, are no longer granted access to the admin API.

For the REST admin API, and any HTML admin web app, bootstrap authentication may be performed either via client certificates, or using another HTTP authentication mechanism (eg basic auth with a manufacturer-supplied bootstrap password).

### Tethered iot-device

A tethered iot-device uses bus-specific access control mechanisms for access to the JSON-RPC interface. For example, in a Bluetooth-attached iot-device, a proxy would first have to pair with the device before having access to the JSON-RPC port.

For a tethered iot-device, the bootstrap process involves configuration of the device by the proxy via the JSON-RPC admin API. At all times the proxy only permits access to the admin API (whether by JSON-RPC or the REST API) to peers belonging to the same Personal Zone. During bootstrap, a proxy does not forward any non-admin API requests from external peers.

## Firmware update requirements

We do not propose to directly address firmware update requirements within the picosec project.

There are three common mechanism

* Proprietary HTTP post: such as typcially supported by routers and other consumer hardware
* OMA firmware management object <http://technical.openmobilealliance.org/Technical/technical-information/release-program/current-releases/fumo-archive>
* Broadband forum RT-069 <http://en.wikipedia.org/wiki/TR-069>
  + <http://www.broadband-forum.org/technical/download/TR-069.pdf>

# Enrolment

## Introduction

Enrolment is the progress of creating secure associations between Picosec entities (ie between IOT devices, IOT devices and IOT applications, or other entities). This section defines the basic requirements and constraints that must be satisifed, and defines the mechanisms by which enrolment is performed in Picosec.

Webinos dealt with enrolment - which in that context specifically relates to creating an association between a device and its owning hub - using one of a number of available authentication procedures, followed by a certificate exchange. Different authentication procedures apply based on the nature of the devices, and the nature of the “introduction” between their owners, but essentially these mechanisms all depend on the devices having some human input and output capability for input and recognition of codes, PINs, etc.

For IoT devices, there are several other problems in the provisioning and management of device networks:

* Small devices have no natural user interface for configuration that would be required for the installation of shared secrets and other security-related parameters. Typically, there is no keyboard, no display, and there may not even be buttons to press. Some devices may only have one interface, the interface to the network.
* Manual configuration is rarely, if at all, possible, as the necessary skills are missing in typical installation environments (such as in family homes).
* There may be a large number of devices, possibly in inaccessible locations. Configuration tasks that may be acceptable when performed for one device may become unacceptable with dozens or hundreds of devices.
* Network configurations evolve over the lifetime of the devices, as additional devices are introduced or addresses change. Various central nodes may also receive more frequent updates than individual devices such as sensors embedded in building materials.

Accordingly a new range of procedures will be necessary, underpinned by new notions of device identity and addressability. This section defines how these objectives are met in Picosec.

## Motivating security requirements

Although the following list is not exhaustive, it provides a set of constraints that must be satisfied by a solution.

1. An IoT device must be capable of being reliably authenticated when it connects to a network, irrespective of whether this is done locally or remotely.
2. A user must be able to enrol the IoT device onto the device securely and easily.
3. A user must be authenticated and authorised to enrol an IoT device on the network.
4. Encryption keys must not be transmitted in plaintext.
5. Credential data such as WiFi passwords must be encrypted.
6. Any other sensitive data such as tokens and session IDs must be encrypted.
7. Sensitive data used in device and user authentication must be secured from modification and manipulation attacks.   
   There must be a provision for data origin authentication (message authentication) of data used for enrolment of the device and user.
8. Sensitive data used in device and user authentication must be secured against replay to protect from spoofing attacks.
9. Sensitive data such as credentials, keys and tokens must be encrypted in storage on the IoT devices and the device management application.
10. The IoT device mut be bound to the user in some way.

## Device identity

Picosec entities are identified in the architecture using self-generated secure identities, similar to Cryptographically Generated Addresses (CGAs)[RFC3972](http://www.ietf.org/rfc/rfc3972.txt) or Host Identity Tags (HITs) [RFC5201](http://www.ietf.org/rfc/rfc5201.txt). That is, we assume the following holds:

I = h(P|O)

where I is the secure identity of the device, h is a hash function, P is the public key from a key pair generated by the device, and O is optional other information (which might include the MAC address of whatever network te device is attached to). The private key from the keypair is held securely on the device; if the device has the hardware capability, this can be in a secure element that performs cryptographic operations using that key.

Picosec does not mandate specific algorithms for the signature or hash algorithm in the architecture, but choices of algorithm are made in Picosec profiles for specific applications. Elliptic Curve algorithms are noted for being amenable to efficient implementation on constrained processors, in comparison with equivalent-strength RSA.

## Bilateral enrolment

A simple bilateral enrolment is a procedure occurring between two Picosec IoT entities (eg two devices, or a device and an application). Enrolment is simply a bilateral key agreement that can occur over any communication channel existing between the entities, or may be out of band (eg via RFID).

Enrolment procedures address two separate issues:

* authentication of the participating entities;
* key agreement.

Authentication of the particpating entities is possible by the following procedures (specified in greater detail in later specifications):

* application level/manual/none: in this case, there is no explicit authentication, but device authentication is implicit. For example, 802.15.4-connected entities might be mutually visible in an environment where it is known that no other devices are present. Alternatively, key agreement may proceed between locally disovered entities, with application-level procedures occurring after key agreement to authenticate the device and confirm the enrolment.
* numeric comparison: if both devices have a display and at least have a Yes/No key. This method displays a numeric code on each device. The user should compare the numbers to ensure they are identical. If the comparison succeeds, the user(s) should confirm the operation on the device(s) that can accept an input. This method provides MITM protection, assuming the user confirms on both devices and performs the comparison properly.
* passkey entry: this method may be used between a device with a display and a device with numeric keypad entry (such as a keyboard), or two devices with numeric keypad entry. In the first case, the display is used to show a numeric code to the user, who then enters the code on the keypad. In the second case, the user of each device enters the same number. Both of theses cases provide MITM protection.
* out of band: an out of band procedure (eg NFC/RFID transfer of identity) is used to authenticate the peering entities.

Key agreement is by using a key agreement protocol such as ECDH where the entities’ keypair is a set of EC params.

Implicit key agreement is where each party’s params are used, each having knowledge of the other party’s public params, to create a canonical DH key, without any information other than public keys being exchanged.

A unique key agreement is where a random nonce is generated by the initiating party and communicated to the other party, and each uses this to create a shared secret key. Unique key agreement is used in situations where it is desirable that the key is not implicitly available based on entity public keys alone - for example if the key is only to be recreated following user entry of a PIN.

The result of bilateral enrolment is a shared secret key that is used in subsequent procedures between the entities.

## Enrolment to IoT hub

An **IoT hub** is an entity that operates as the “owner” of one or more IoT entities. The role of the IoT hub is analogous to the role of **Personal Zone Hub**in Webinos. A hub is identifiable and locatable: an IoT hub has a stable URI, and various services would be expected to be served at that location by the hub’s provider.

A hub provides a means to discover its attached enrolled IoT devices, and provides a means to address those devices, eg as:

GET <hub URL>/devices/<device id>

The IoT entity -> hub relationship is usually exclusive; ie an IoT entity is only associated with a single IoT hub at any time.

An IoT hub is an IoT entity in its own right, and therefore has a keypair and identity. Enrolment of a device with the IoT hub is a simple bilateral enrolment operation, performed using any of the available methods.

A hub may enrol with another IoT entity (such as an IoT app) directly, or may enrol any of its child devices via delegated enrolment (see below).

Entities that are enrolled to the hub are issued with a **membership ticket**, which is an artifact signed by the hub, containing the member entity’s identity.

### Delegated enrolment

This is the stuation where an IoT entity (typically a device) is enrolled to another entity (eg an app) via a mediating IoT hub.

The authentication of the device, in this case, is by authentication of the IoT hub, which in turn attests to the authenticity of the owned entity. Key agreement is performed by independent bilateral exchanges between the hub and each entity. This means that an enrolment between a hub-attached device and any other entity can take place without the device being online; it can obtain the necessary parameters to construct the key for itself when it first requires it.

A hub therefore acts as a grouping mechanism for child entities, being able to mediate global discovery, global addressing and enrolment, without itself mediating actual service invocation.

## Aggregate entities

An IoT hub is a useful mechanism for grouping entities but it forces a hierarchical structure on participating entities. It also requires a stable URL, which requires an online provider. There is also a desire that Picosec supports a fully federated peer-to-peer model whilst retaining the ability for entities to operate in groups to ensure management and connectivity remain scalable.

The **IoT aggregate** is an entity that performs this role. An aggregate is purely a **virtual entity**: it has an IoT identity, but does not necessarily reside on a physical device, and it does not necessarily have a cloud presence or URL.

An IoT aggregate represents a group of peer entities of without implying any hierarchy of status between those entities. An aggregate serves to provide logical grouping and, in the same way as for an IoT hub, enables delegated enrolment between its member entities and any other IoT entity.

An IoT aggregate may have a physical presence in an aggregator device (ie a bridge or gateway, as might exist between a wireless mesh network and the internet). Such a device is visible as an IoT entity on the network in its own right, and it may export services that permit the member devices to be discovered and addressed. Although the aggregator might not typically have a stable URL, or even a stable IP address, it will nonetheless be discoverable using local means - eg [mDNS](http://tools.ietf.org/html/rfc6762) - so local entities such as local apps can find the aggregator and thence each of its member devices.

### Self aggregating devices

A further possiblity is a group of **self-aggregating devices**: a collection of devices that can each peform the role of aggregator from time to time. This mode of operation means that an aggregate can exist without any dedicated hardware.

As with hub enrolment, an entity that is a member of an aggregate is issued with a membership ticket; this ticket contains the information that any other entity needs in order to perform delegated enrolment for that member with any third (external) entity. All members of the aggregate have visibility of all member membership tickets.

Management of the IoT aggregate’s keypair may by by one of the following patterns.

### Aggregation using shared credentials

In this model, each member of the aggregate has access to the keypair for the aggregate. This model can apply only when the following conditions are satisfied:

* all members are trusted equally and are authorised to perform delegated authentication and key agreement for all other members;
* compromise of aggregate credentials from member devices is considered acceptable. This includes the requirement that member devices are not re-used in any subsequent setting without being restored to a state in which the aggregate credentials are not present.

### Aggregation via an aggregate manager

In this model, an entity (such as an IoT app) acts as the manager of the aggregate entity’s credentials. This is akin to having an app acting as a wallet or locker for credentials. The app is involved in issueing new membership tickets when new devices are enjoined to the aggregate.

The manager app does not have any role in the ongoing operations of the aggregate, and does not mediate access between any member entity and any client IoT entity.

# Secure connection

## Overview

Picosec is intended to address a very wide range of use cases, and solutions will exist that use a broad range of technologies and architectures; we do not aim to create a “one size fits all” approach. To this end, Picosec will establish a solution elements - for example protocols, procedures and formats for key agreement - that are applicable within different architectures. These solution elements, delivered as a combination of specifications and open source code for reference implementations, will be capable of integration into different solutions and platforms.

On top from these solution elements, Picosec provides a complete solution architecture that targets a broad range of use cases. We do not claim that this architecture is suitable for every scenario, but it specifically aims to confront, directly, the most significant challenges of the next generation of IoT applications. It is strongly influenced by Webinos ideas - and includes Webinos-defined APIs and protocols in some elements - but re-casts these for IoT applications with the the attendant constraints relating to trust, communications, hardware capability and scale. The architecture extends existing IOT solutions by addressing:

* federated systems where there is heterogeneity in device ownership and trust, instead of systems in which end-devices and the systems that talk to them operate under a single authority;
* extensible systems where the services offered by a device are dynamic, discoverable and negotiable instead of being fixed;
* systems in which services are exchanged over disparate, changing, and independently owned networks, instead of having dedicated and co-managed interconnection.

These, we think, are the key characteristics of something we could describe as an “Internet of Things” - or even eventually a “Web of Things” - instead of a collection of independent systems having only a bridgehead to the internet.

## Key solution components

The primary elements of the baseline Picosec solution architecture are:

* **CoAP**. CoAP is the protocol that allows a range of services, each with their different interaction and API styles, to be supported over unreliable networks. It is the natural successor to the Webinos “JSON-RPC + reliable transport” model for the IoT environment. CoAP permits both symmetric and asymmetric interaction and has the range of features needed to satisfy the discoverability and extensibility requirements of Picosec use cases.
* **DTLS/DICE**. A key element of our approach is that service exchange must be possible over untrusted and independently owned networks or subnetworks and security is provided end-to-end over those networks. This requires the existence of an internetworking protocol - UDP - and a end-to-end security protocol over that. DTLS meets this need, and DICE provides profiles, and certain specific additional mechanisms, to allow those to operate effectively in constrained environments.
* **Efficient session state handling**. One of the most difficult aspects of running DTLS in very constrained environments is the computational resource-intensivity of public key operations, and the communications resource-intensivity of session negotiation. Picosec specifies the use of long-lived session parameters, determined on a per-service-binding basis, and shareable in a standardised format using a RFC5077 session resumption ticket. [Extended DTLS Session Resumption](http://tools.ietf.org/html/draft-hummen-dtls-extended-session-resumption-00) is adopted to enable symmetric use of session resumption features.
* **Flexible identification and authentication regimes**. The protocols described above can based on pre-shared keys for environments that operate under a single authority and are homogeneously trusted, and [raw public keys](http://tools.ietf.org/html/draft-ietf-tls-oob-pubkey-11) for heterogeneous environments, or any other application where public-key-based identities are required.
* **Service-level policy control**. Picosec defines standardised policy handling formats and mechanisms, with policy decision and enforcement occurring at the service binding stage (as opposed to service invocation in Webinos). This is allied to the creation of per-binding session credentials.

Requirements for each of these elements are given in the sections that follow.

### CoAP

[CoAP](https://datatracker.ietf.org/doc/rfc7252/) is the baseline protocol for interactions between entities in the Picosec architecture. This facilitates an HTTP-style of interaction over a datagram transport, and contains many other changes to ensure its suitability for constrained devices and networks. Furthermore, CoAP is symmetric - that is, there is no enduring distinction between clients and servers - which provides further flexibility in the construction of complex architectures.

In later porject stages, detailed profiles will be constructed detailing which CoAP features are required of all Picosec implementations, and which are optional. The intention, however, is to ensure that the set of mandatory features is as small as possible.

In the baseline architecture, access control between devices and services is mediated by service binding, and transport-level connection parameters are made available to enable service invocation on the basis of that access control function. This means that there is no further requirement for authentication and authorisation at the CoAP level, except where more fine-grained policies need to be implemented. The Picosec project will track the ongoing work of [Authentication and Authorization for Constrained Environments (ace)](https://datatracker.ietf.org/wg/ace/documents/) and those recommendations may be taken up as that work matures.

There is a natural translation of REST APIs from HTTP to CoAP; this is further refined for constrained devices in the [Constrained RESTful Environments (core)](https://datatracker.ietf.org/wg/core/documents/) work. These recommendations are adopted for Picosec.

Webinos APIs, where applicable to Picosec, are exposed as CoAP REST APIs; the specific bindings are defined in the context of each particular API.

Sensor event delivery, and event delivery for other discrete streams, is supported in Picosec via the [Observe extension](https://datatracker.ietf.org/doc/draft-ietf-core-observe/).

Picosec does not mandate navigable [HATEOS](http://en.wikipedia.org/wiki/HATEOAS) APIs due to the general impact of expanding response payloads in a constrained environment; however, their use is encouraged and where they are supported the use of [RFC6690](https://datatracker.ietf.org/doc/rfc6690/?include_text=1) is required.

### DTLS/DICE

Picosec supports CoAP running over UDP directly and over [DTLS](http://tools.ietf.org/html/rfc6347). [UDP](http://tools.ietf.org/html/rfc768) is acceptable for situations where there is no confidentiality, authentication of authorisation requirement. CoAP may also run directly over a subnetwork layer, without IP, where IP is not available, assuming that there is an aggregator that can expose the attached devices to the IP network.

In all other cases, DTLS is required. DTLS contains a very large number of options and parameters and options need to be carefully chosen to render it suitable for very constrained environments. The [DICE](https://datatracker.ietf.org/wg/dice/documents/) effort aims to define profiles for DTLS that are suitable for IoT environments.

The principal measures adopted to optimise DTLS for Picosec are:

* use of pre-shared keys for authentication. Where Picosec entities have a bilateral relationship through enrolment, the resulting shared key can be used for mutual authentication as defined initially in [RFC4279](http://tools.ietf.org/html/rfc4279) and updated for DTLS in the DICE specification.
* use of pre-shared keys as session keys, for instances where this lightweight approach is acceptable. The cipher is TLS\_PSK\_WITH\_AES\_256\_CBC\_SHA. The primary issue here is the lack of forward secrecy, so it is only recommended for situations where there is no confidentiality or authorisation requirement.
* use of DH-negotiated session keys for forward secrecy. The cipher here is TLS\_DHE\_PSK\_WITH\_AES\_256\_CBC\_SHA.
* Picosec also supports raw public keys via the [out of band pubkey extension](http://tools.ietf.org/html/draft-ietf-tls-oob-pubkey-11) where an out-of-band mechanism is used for exchange of public keys in lieu of enrolment.

### Efficient session state handling

Access to services that have an authorisation requirement is subject to a binding-specific session key; it is not acceptable to use the bilateral entity key. Each time an entity binds to a specific service on another entity, an authorisation check is performed, and a new session key is negotiated. That key is used for all DTLS traffic for that service invocation.

Thus, session state is preserved and sessions are continued on a long term basis (until eother entity wishes to restart), instead of being renegotiated on each access episode.

Picosec adopts the convention that such session state is persisted in a standardised format using a [RFC5077](http://tools.ietf.org/html/rfc5077) session resumption ticket. [Extended DTLS Session Resumption](http://tools.ietf.org/html/draft-hummen-dtls-extended-session-resumption-00) is adopted to enable symmetric use of session resumption features.

In principle, this arrangement also permits access authorisation and key negotiation to be performed out of band, with session resumption tickets distributed to the peers out of band. Service invocation can then proceed without any further initialisation of state. Use cases for this option are still under investigation.

### Flexible identification and authentication regimes

As mentioned above, both pre-shared keys and out-of-band raw public keys can be used.

PKIX certificate-based keys are supportable but are not considered to be widely applicable for constrained devices and are thus not the primary target.

### Service-level policy control

The Webinos architecture, itself originating from BONDI, defined a rich access control framework wherein arbitrarily complex policies could be authored, and amended in-place, for access by clients to services. That generality was motivated by the wide range of privacy and security concerns that are evident in Webinos applications, and its realistion was feasible on the type of platform targeted by Webinos.

For Picosec, that level of generality is neither required nor feasible. The first-generation Picosec functionality is based on bind-time application of access control policy. Subsequent versions of the architecture may support in-service access controls, but the requirement for these is still not certain.

Picosec does not mandate an interoperable policy format, and does not mandate any specific policy configuration or decision functionality; this is too broad a problem given the nature of Picosec applications. The Picosec reference implementation will include a specific reference implementatin of a Policy Decision Point (PDP) but this is not part of the standard.

Picosec simply requires that policy enforcement is applied by granting or denying access to a session key for the client attempting a bind. Picosec entities that export services are required to demand a negotiated session key (instead of using the bilateral enrolment key) for client sessions.

## Alternative architectures

Although the primary target architecture is based on the components listed above, there are instances where other architectures are used, either for legacy or interoperability reasons, or simply because they are better suited to the specific requirements or constraints of the application. In these situations, Picosec elements may still be usable within the overall context of the architecture, and there might also be a question of interoperability issue between those systems and baseline Picosec systems.

### TLS

There will, or course, be many applications where TLS is supportable; this depends obviously on the supportability of TCP. Where a network supports IP then TCP is clearly supportable in principle; a decision to use TCP or UDP is really an optimisation decision, based on network characteristics and the reliability requirements of the application running over it.

Picosec supports TCP/TLS-based architectures by:

* **defined profiles**. Specific standard profiles are identified for Picosec applications running over TLS. These cover both the PSK use-case and raw public key case.
* **interoperable session state encapsulation**. The Picosec-defined format based on RFC5077 supports sessions over TLS or DTLS.
* **symmetric session resumption**. The session resumption type negotiation is supported to allow TLS session resumption without client-side state.

These measures ensure that Picosec service discovery, binding, access control and policy mechanisms operate equivalently and interchangeabley between TLS and DTLS.

### PKIX

There will be systems that use PKIX mechanisms for public key distribution and trust instead of raw public keys. Picosec defines minimum profiles to enable these systems to use Picosec mechanisms, and interwork with baseline Picosec systems.

### Application-level security

Some systems will implement end-to-end application-level security instead of transport-level security. This may be appropriate where the sensitivity of the application does not require protection of lower protocol layers, or where the system operates over networks that have sufficient intrinsic security properties to remove the need for transport-level security. Picosec key agreement mechanisms, and session key exchange format, may be used with such systems.

### MQTT

MQTT is a popular protocol for interconnection of IoT devices, particularly sensor devices, with consuming systems. It is a publish/subscribe (“pubsub”) protocol that operates over a reliable transport (ie TCP or TLS). Picosec may be applied to MQTT-based systems in two ways:

* by providing key exchange and device authentication support where MQTT operates over an insecure transport using application-level security. MQTT data payloads may be encrypted using keys that were created or distributed using Picosec mechanisms.
* by providing a lightweight implementation of TLS, to allow MQTT to run with transport-layer security. Picosec TLS profiles and key/session state mechanisms can be used in support of that TLS layer.

In the special case of MQTT-SN, the first of these applies where the MQTT-SN protocol operates directly over a subnetwork without an internetworking layer.

# Authorisation requirements

The base authorisation model is that a Policy Decision is made at service bind time, resulting in a session key being issued; that session key is used for subsequent sessions for that service between the requesting and serving entities. Policy Enforcement is by the serving entity in that any policy-controlled service is only accessible over connections that have a corresponding session key.

The limitations of this basic approach are that the Policy Decision Point only has the identity of the requesting entity on which to make a decision; this is too limited for most use-cases. In many, the identity of the requesting entity is just meaningless; in others, even if in principle the serving entity could know enough about each individual requesting identity, it is not scalable to expect this to happen for the massive numbers of potential legitimate requestors.

There are two proposals for extending the basic scheme:

* **conferring access rights to aggregates or hubs**: a serving entity might, in addition to knowing the identity of the individual requesting entity, know that entity’s membership of aggregates or its association with an IoT hub. A Polucy Decision Point could then make a decision based on that information. This would address the scalability issue in part, where access rights genuinely align with aggregate membership or the hierarchy of hub ownership. Simply managed access “zones” can be established using this method.
* **capability-based access control**: a requester obtains a capability or token - by a method yet to be specified - and presents this capability to the serving entity. Here again there are multiple possible models; a capability might be presented at bind time, and following a Policy Decision a session key is issued; or a capability might be presented in combination with a session key at invocation time. Each option has different management and scalability implications which need to be studied further.

The format of capabilities needs to be studied; in particular whether or not they have an open format or are simply opaque to all except the issuer, and whether they are literal - ie are self-describing and can be acted upon without any need for a look up to establish their validity or content - or they are simply references to a queryable database whose entries contain the capability details.

The requirements for the security of storage for credentials also need to be determined. In general, however, capabilities would be comparatively short-lived so the level of protection needed would be less than long-lived credentials.

# Security: permission granting

define the model for device/service discovery and invocation for both driver and full IOT model

## Policy requirements

Requirement: A policy description language must be provided with can express policy constraints flexibly

Requirement: The policy language must be able to restrict access to APIs

Requirement: The policy language must be able to restrict access named devices

Requirement: The policy language must be able to restrict devices owned by people

Requirement: The policy language must be able to restrict access based on calling application

Requirement: The policy language must be extendable to new APIs

Requirement: The policy language must be able to restrict based on API specific credentials (e.g. folder name or time of day)

Requirement: The policy language must be efficient to parse and store

Requirement: The discoverability of an entity and services must be protectable by policy

Requirement: Policy must be syncronisable and manageable – an interoperable definition of policy must exist

Requirement: The Policy description framework must be extensible by third parties.

Requirement: Policy management must be possible through easy to use API

## Policy scenarios

We should consider two policy control scenarios that relate to the Hub model and peer to peer model respectively. The difference between the two is as follows

### Peer to Peer based policy:

This model is close to the bluetooth model for security. Each device creates one or more relationships with other IOT-applications (or possibly devices). This peer to peer relationship is represents both:

* trust in identity: the device-app know and trust each other (in all probability by a trusted key exchange)
* permission to use services. presentation of the key therefore is granting an application permission to use the service, where of course trust in the presented applications identity is presumed

Importantly, for the peer to peer model, the device and only the device is responsible for policy enforcement.

No policy synchronisation is required and the permissions granted to a particular key are interpreted by (and can be changed dynamically) by the iot-device

### Hub based security

In a hub based security model an intermediate hub takes responsibility for policy control and enforcement. When a device joins a hub two things happen

1. An identity is issued by the hub to the iot-device. This is a strong trusted identity which is capable of being used to initiate peer to peer trust relationships
2. The hub is granted the ability to overwrite policy rules. This is essential if we which to manage many IOT devices as one. For example grouping lightbulbs together into a single unit and writing the permissions to that devices

A third facet of hub model, not directly related to policy, is that the hub can act as a messaging proxy for the device.

### Strengths and weakness of Hub vs Peer based policy model

* Peer to peer
  + (plus) no infrastructure/intermediary needed
  + (plus) simple
  + (minus) does not scale well to many devices
* Hub based
  + (plus) scales well to many devices
  + (plus) usable for end users
  + (minus) infrastructure deployment needed
  + (minus) more complex IOT stack

# Policy Specifications

## Hub based policy specifications

We have very well formed specification for handling hub based policy.

It remains an engineering challenge to see how small we can implement this stack

* Policy Description
  + [BONDI\_Architecture\_and\_Security](http://picosec.org:8080/specs/BONDI_Architecture_and_Security)
  + [BONDI\_Architecture\_and\_Security\_Appendices](http://picosec.org:8080/specs/BONDI_Architecture_and_Security_Appendices)
  + [webinos policy](http://picosec.org:8080/specs/webinos%20policy)
* Certificate enrolment
  + [webinos enrollment](http://picosec.org:8080/specs/webinos%20enrollment)

This policy model is declarative and syncronisable between different iot-devices.

The current implementations of this security model are too heavy for anything other than full hub-driver deployments

## Peer 2 peer based policy

The peer to peer model will require extensive experimentation and refinement.

It will consist principally of a key exchange protocol such as <http://en.wikipedia.org/wiki/SPEKE_(cryptography>) which is intern based on Diffie Helman key exchange <http://en.wikipedia.org/wiki/Diffie-Hellman_key_exchange>.

All subsequent requests and responses are then hashed with the key in order to assure both identity and trust.

Optimisations may be possible using an intermediate session key.

A simple key hieracy will be constructed where

* root key is used to overwrite all other keys
* identity key: is used to establish device identity only, there may be
* capability key: is used to represent both device identity and the permissions granted to a client

In the peer to peer model the iot-device has total autonomy over the interpretation of the capability key.

In this model policy neither declarative nor syncronisable

# Intermediate model

There is an outstanding question as to whether an intermediate model will be required.

This model need the simplicity of the capability model, but because the interpretation of capability needs to be shared between multiple iot nodes, the model will have to be syncronisable and therefore describable.

# Data management

IOT devices are not always connected

IOT devices need to be able to record and cache data samples to upload, or be accessed later.

Using the remove invocation capabilities it is possible to remotely query the device from the IOT application if a suitable interfaces is provided.

We propose to use a simple JSON friendly local DB API to faciliate this remote querying.

The draft specification is based on the webinos DB API which in tern is based on the MongoDB <http://www.mongodb.org/> /TingoDB<https://github.com/sergeyksv/tingodb> schemes

We are evaluating an alternative comprising the CouchDB <http://couchdb.apache.org/> /PouchDB <http://pouchdb.com/> system.

At an implementation level it needs to determined whether a low footpring C based implementation of these external interfaces is possible

Also integral to more sophisticated IOT data management is the possibly for remotely manageable micro interpreters based for example on Tiny.js<https://code.google.com/p/tiny-js/>

## Data Specification

<http://www.tingodb.com/info/api/>

# TingoDB

**TingoDB** is an embedded JavaScript in-process filesystem-backed database upwards compatible with MongoDB at the API level.

Upwards compatible means that if you build an app that uses functionality implemented by TingoDB you can switch to MongoDB almost without code changes. This greatly reduces implementation risks and give you freedom to switch to a mature solution at any moment.

As a proof for upward compatibility, all tests designed to run against both MongoDB and TingoDB. Moreover, significant parts of tests contributed from MongoDB nodejs driver projects and are used as is without modifications.

For those folks who familiar with the Mongoose.js ODM, we suggest to look at [Tungus](https://github.com/sergeyksv/tungus), an experimental driver that allows using the famous ODM tool with our database.

For more details please visit [http://www.tingodb.com](http://www.tingodb.com/)

# Usage

npm install tingodb

As stated, the API is fully compatible with MongoDB. The only differences are the initialization and getting the Db object. Consider this MongoDB code:

var Db = require('mongodb').Db,

Server = require('mongodb').Server,

assert = require('assert');

var db = new Db('test', new Server('locahost', 27017));

var collection = db.collection("batch\_document\_insert\_collection\_safe");

collection.insert([{hello:'world\_safe1'}

, {hello:'world\_safe2'}], {w:1}, function(err, result) {

assert.equal(null, err);

collection.findOne({hello:'world\_safe2'}, function(err, item) {

assert.equal(null, err);

assert.equal('world\_safe2', item.hello);

})

});

The same example using TingoDB will be as follows:

var Db = require('tingodb')().Db,

assert = require('assert');

var db = new Db('/some/local/path', {});

// Fetch a collection to insert document into

var collection = db.collection("batch\_document\_insert\_collection\_safe");

// Insert a single document

collection.insert([{hello:'world\_safe1'}

, {hello:'world\_safe2'}], {w:1}, function(err, result) {

assert.equal(null, err);

// Fetch the document

collection.findOne({hello:'world\_safe2'}, function(err, item) {

assert.equal(null, err);

assert.equal('world\_safe2', item.hello);

})

});

As you can see, the difference is in the require call and database object initialization.

#### require(‘tingodb’)(options)

In contrast to MongoDB, the module require call will not return a usable module. It will return a function that accepts configuration options. This function will return something similar to the MongoDB module. THe extra step allows for passing some options that will control database behavior.

##### nativeObjectID: true|false Default is false

Doing some experimentation we found that using integer keys we can get the database to work faster and save some space. Additionally, for in-process databases there are almost no drawbacks versus globally unique keys. However, at the same time, it is relatively hard to keep unique integer keys outside of the database engine, so we made it part of the database engine code. Generated keys will be unique in the collection scope.

When required, it is possible to switch to BSON ObjectID using the configuration option.

##### cacheSize: integer Default is 1000

Maximum number of cached objects per collection.

##### cacheMaxObjSize: integer Default is 1024 bytes

Maximum size of objects that can be placed in the cache.

##### searchInArray: true|false Default is false

Globally enables support of search in nested arrays. MongoDB supports this unconditionally. For TingoDB, search in arrays when there are no arrays incurs a performance penalty. That’s why this is switched off by default. Additionally, and this might be better a approach, nested arrays support can be enabled for individual indexes or search queries.

To enable nested arrays in individual indexed, use “\_tiarr:true” option.

self.\_cash\_transactions.ensureIndex("splits.accountId",{\_tiarr:true},cb);

To enable nested arrays in individual queries for fields that do not use indexes, use “\_tiarr.” to prefix field names.

coll.find({'arr.num':10},{"\_tiar.arr.num":0})

#### new Db(path, options)

The only required parameter is the database path. It should be a valid path to an empty folder or a folder that already contains collection files.

# Dual usage

It is possible to build applications that will transparently support both MongoDB and TingoDB. Here are some hints on how to do that:

* Wrap the module require call into a helper module or make it part of the core object. This way you can control which engine is loaded in one place.
* Use only native JavaScript types. BSON types can be slow in JavaScript and will need special attention when passed to or from client JavaScript.
* Treat ObjectID just as a unique value that can be converted to and from String regardless its actual meaning.

Example below (please see the three files).

###### engine.js - wrapper around TingoDB and MongoDB

var fs = require('fs'),db,engine;

// load config

var cfg = JSON.parse(fs.readFileSync("./config.json"));

// load requestd engine and define engine-agnostic getDB function

if (cfg.app.engine=="mongodb") {

engine = require("mongodb");

module.exports.getDB = function () {

if (!db) db = new engine.Db(cfg.mongo.db,

new engine.Server(cfg.mongo.host, cfg.mongo.port, cfg.mongo.opts),

{native\_parser: false, safe:true});

return db;

}

} else {

engine = require("tingodb")({});

module.exports.getDB = function () {

if (!db) db = new engine.Db(cfg.tingo.path, {});

return db;

}

}

// Depending on engine, this can be a different class

module.exports.ObjectID = engine.ObjectID;

###### sample.js - Dummy usage example, pay attention to comments

var engine = require('./engine');

var db = engine.getDB();

console.time("sample")

db.open(function(err,db) {

db.collection("homes", function (err, homes) {

// it's fine to create ObjectID in advance

// NOTE!!! we get class through engine because its type

// can depends on database type

var homeId = new engine.ObjectID();

// but with TingoDB.ObjectID righ here it will be negative

// which means temporary. However it's unique and can be used for

// comparisons

console.log(homeId);

homes.insert({\_id:homeId, name:"test"}, function (err, home) {

var home = home[0];

// here, homeID will change its value and will be in sync

// with the database

console.log(homeId,home);

db.collection("rooms", function (err, rooms) {

for (var i=0; i<5; i++) {

// it's ok also to not provide id, then it will be generated

rooms.insert({name:"room\_"+i,\_idHome:homeId}, function (err, room) {

console.log(room[0]);

i--;

if (i==0) {

// now lets assume we serving request like

// /rooms?homeid=\_some\_string\_

var query = "/rooms?homeid="+homeId.toString();

// dirty code to get simulated GET variable

var getId = query.match("homeid=(.\*)")[1];

console.log(query, getId)

// typical code to get id from external world

// and use it for queries

rooms.find({\_idHome:new engine.ObjectID(getId)})

.count(function (err, count) {

console.log(count);

console.timeEnd("sample");

})

}

})

}

})

})

})

})

###### config.json - Dummy config

{

"app":{

"engine":"tingodb"

},

"mongo":{

"host":"127.0.0.1",

"port":27017,

"db":"data",

"opts":{

"auto\_reconnect": true,

"safe": true

}

},

"tingo":{

"path":"./data"

}

}

###### Console output running on TingoDB

-2

13 { \_id: 13, name: 'test' }

{ name: 'room\_0', \_idHome: 13, \_id: 57 }

{ name: 'room\_1', \_idHome: 13, \_id: 58 }

{ name: 'room\_2', \_idHome: 13, \_id: 59 }

{ name: 'room\_3', \_idHome: 13, \_id: 60 }

{ name: 'room\_4', \_idHome: 13, \_id: 61 }

/rooms?homeid=13 13

5

sample: 27ms

###### Console output running on MongoDB

51b43a05f092a1c544000001

51b43a05f092a1c544000001 { \_id: 51b43a05f092a1c544000001, name: 'test' }

{ name: 'room\_3',

\_idHome: 51b43a05f092a1c544000001,

\_id: 51b43a05f092a1c544000005 }

{ name: 'room\_2',

\_idHome: 51b43a05f092a1c544000001,

\_id: 51b43a05f092a1c544000004 }

{ name: 'room\_1',

\_idHome: 51b43a05f092a1c544000001,

\_id: 51b43a05f092a1c544000003 }

{ name: 'room\_0',

\_idHome: 51b43a05f092a1c544000001,

\_id: 51b43a05f092a1c544000002 }

{ name: 'room\_4',

\_idHome: 51b43a05f092a1c544000001,

\_id: 51b43a05f092a1c544000006 }

/rooms?homeid=51b43a05f092a1c544000001 51b43a05f092a1c544000001

5

sample: 22ms

# Compatibility

We maintain full API and functionality compatibility with MongoDB **BUT** only for what we implemented support. I.e. if we support something it will work exactly the same way, but some features are not yet supported or support is limited.

* Search, almost all clauses. Indexes are used to increase search speed and sorting.
* Map reduce, almost all
* Grouping, almost all
* Collection, almost all methods
* Cursor, almost all methods
* Indexes, no support for compound indxes, only single-field indexes are supported. Full text search is also not supprted
* GridFS, no support
* Feature X, might be :)

# Miscellaneous requirements

These requirements don’t naturally fall in any of the existing sections, but are worth recording regardless.

## Device identity requirements

Requirement: Strong identity of device, users must be supported, ideally with PKI certificates

Requirement: A process of enrolment is required where a device is bound to a user

Requirement: The process of binding a device to a user must be revocable

## Application identity and integrity

Requirement: Applications should support a packaging mechanism that an detect corruption and therefore support integrity capability

Requirement: Application packaging mechanism must be space efficient

Requirement: Application packaging mechanism must be easy to parse

Requirement: Applications must support strong identities, ideally through PKI certificates

Requirement: Applications must support certificate chaining to help establish provenance of an application

Requirement: Application must support certificate chaining to digitally model secure workflow

## Internet of Things Requirements

Requirement: The solution must provide real time data extraction APIs through RESTful or alternative strategies, not just bulk data uploads.

Requirement: The solution must allow for “secure” and “interoperable” integration of Internet of Things like sensors to future proof for advanced SmartCity deployment scenarios.

Requirement: The solution must allow for “secure” and “interoperable” integration of Internet of Things like actuators to facilitate intelligent distributed control of IOT devices.

Requirement: The IOT integration mechanism MUST have model that can be post installed to existing kit, i.e. provide legacy interoperability mode.

Requirement: The sensor integration model must be fit for purpose for IOT providing solution that will work even in non IP scenarios.

Requirement: The sensor integration model must be easy to program for both remotely and locally.

## Application execution /Process requirements

Requirement: There should be two application types – UI applications and background applications

Requirement: UI applications which render graphics to the screen on user insanitation

Requirement: Background applications should (optionally) auto start on device start up and run as background processes

Requirement: Applications should be in protected memory space from one another to avoid cross application corruption

Requirement: Application start up time must be less than 100 msec

Requirement: Local UI must be provided for managing (terminating) errant applications

Requirement: Remote UI must be provided for trusted parties (policy permitting) to manage applications

## Data handling/ data management requirements

Requirement: The solution must support easy to use data caches for data performance reasons

Requirement: There should be an option to supported Encrypted data at rest, for prime data stores, cached data stores and application data stores

Requirement: Data should be storable in a central accessible store in a variety of different electronic formats, including (but not limited to)

## Synchronisation requirements

Requirement: Synchronisation functions should be supported for at least

* NOSQL data
* Tabular data
* HTML5 Application data
* HTML5 Session store data

Requirement: Synchronisation should be possible between single device instance of data and single server instance

Requirement: Synchronisation must be supported between multiple server instances

Requirement: Synchronisation must be supported between multiple device instances.

Requirement: Synchronisation data types should be categorised as either device global (any application can access) or application specific.

Requirement: Synchronisation should be efficient: communicate minimal delta information only

Requirement: An option should be supported for realtime synchronisation

Requirement: Synchronisation should support b-directional sync, server changes or client changes

Requirement: Synchronisation refreshes should be trigger able from client or server

## Specific API requirements

Requirement: An interoperable secure JavaScript API is required to grant full file read access

Requirement: An interoperable secure JavaScript API is required to grant full file write access

Requirement: An interoperable secure JavaScript API is required to manage file bindings

Requirement: An interoperable secure JavaScript API is required to grant full NFC read access

Requirement: An interoperable secure JavaScript API is required to grant full NFC write access

Requirement: An interoperable secure JavaScript API is required to access geo location information

Requirement: An interoperable secure JavaScript API is required to access device orientation information

Requirement: An interoperable secure JavaScript API is required to create binding to arbitrary sensors

Requirement: An interoperable secure JavaScript API is required to create binding to arbitrary actuators

## Management requirements

Requirement: Administration of users, including how different roles are set up and their permissions configured to perform functions in the solution and the workflow

Requirement: It must be possible to both locally and remotely Configure APIs and API instances

Requirement: Multi factor authentication should be provided to secure end points

Requirement: Lock and wipe of end user device functions should be supported where they may cache sensitive data

Requirement: The systems must support administration delegation functions

Requirement: The systems must support facilities for users to sign up to use the functions of the data store (either as a member of a partner organisation or as an individual), to manage their own profile and preferences in the solution, including saving customised visualisations and functions

Requirement: The systems must support managing the terms and conditions of use of the data store/web wrapper and the ability to moderate users’ behaviour.

Requirement: A user verification process must be in place as part of the registration

Requirement: The system must provide an ability to manage the functions and preferences of the solution, such as uploading data, data visualisations

Requirement: The system must provide an ability to handle configuration and management of metadata

Requirement: The system must provide an ability to te ability to remove/suspend data

Requirement: The system must provide an ability to manage of historic versions of data sets

Requirement: The system must provide an ability to manage data retention periods

Requirement: The system must provide an ability to manage data suppression limits

Requirement: The system must provide an ability to the administration and management of the workflow to upload data

Requirement: The system must provide an ability to monitor the use/load of the interfaces/APIs

Requirement: The system must provide an ability to the ability to suspend or stop interfaces/APIs, to individual systems/users, individual APIs or individual users.

Requirement: The system must produce user activity reports and tools to analyse use of the system

Requirement: The system must provide an ability to manage the space used by the system and the facilities to grow the space of the solution.

## Networking requirements

Networking requirements define the nature of the physical and logical connection between application and service end points.

Requirement: Connections from a device to a remote resource, should, as far as possible share physical resources (ie multiplex through the same IP connection)

Requirement: Connections originating from applications must declare the unique identity of the application making the connection

Requirement: Network transported application identities, should be, Non-repudiable, as far as possible

Requirement: Connections between core runtine and browser UI components should be secure (attestation of end points)

Requirement: End points should support RESTful server interfaces where possible for backward compatibility

Requirement: Network connections should be “lightweight” ie the proportion between payload and messaging metadata should be less that 5%

Requirement: Network connections should be efficeient: ie multiplexing and security infrastructure should not add more than 20ms to round trip messaging