Mediator Architecture Overview
1. Introduction

The Mediator is a system composed of both hardware and software that is collectively used to gather and manipulate data from numerous sources. These sources are typically intelligent machines or sensors, found in virtually any facility, which are otherwise unable to intercommunicate. The Mediator allows communication to occur between these devices, facilitates additional processing, and provides a uniform presentation of this information to users (i.e. web-browsers) and third party applications. The main objectives of this work are as follows:

1) Provide an overview of the Mediator’s hardware platform.
2) Provide an overview of the Mediator’s software platform.
3) Provide an overview of the interfaces and services the Mediator delivers.
4) Provide an overview of how the Mediator’s framework is structured.
5) Provide an overview of XML-RPC in general and how the Mediator uses it in particular.

The intent of this paper is not to provide comprehensive coverage of all the Mediator’s features and functionality, but rather to provide an overview that might incite additional dialogue.

2. Hardware Overview

The Mediator presently runs an AMD Geode™ SC1100 processor at 266 MHz. The SC1100 single-chip processor includes a 32-bit x86-compatible AMD Geode GX1 processor that is targeted towards the creation of x86 based embedded systems. Depending on the intended application, it is shipped with varying RAM (between 32Mb and 512Mb) and Compact Flash (128Mb up to 4Gb) sizes. To meet the diverse “Layer 2” requirements found in buildings, it is equipped with a number of different physical interfaces, including:

(2) solid-state, opto-isolated, single pole normally open relay’s (0-60V, 1.0A AC/2.0A DC)

(4) Dallas 1-Wire ports

(4) Pulse Counter Inputs

(4) RS-485 ports – each capable of binding a distinct communication protocol to it.

(2) RS-232 ports – each capable of binding a distinct communication protocol to it.
(2) Ethernet ports – each capable of binding multiple communication protocols to it.

(1) RJ-11 Phone\Modem port.

(2) USB ports

3. Software Overview

The Mediator’s software is logically divided into a MOE (Mediator Operating Environment) and a Framework. The MOE Richards-Zeta utilizes is based on a 2.4 series Linux kernel and a Red Hat derived “user-space”. Its multi-tasking, sophisticated memory management and robust networking stack allows the Mediator to run multiple services and protocols simultaneously. Where performance is an absolute must, the Linux operating system allows for the creation of kernel modules – pieces of code that can be loaded directly into the kernel on demand. Several kernel modules have been written so that the Mediator might deal with different building automation protocols that have strict performance (ie. token passing) requirements and to work with on-board hardware (ie. relays, UART’s, etc.). Also included in the MOE are various user-space applications (ie. ssh, openvpn, etc.) that add substantial value to the overall system.

At the core of the Mediator sits the Framework or MPX (Multi-Protocol eXchange). From a programming perspective, the Mediator had the following design goals:

- Introspective – allow “poking around” a running system.
- Non-monolithic – allow for incremental changes in the field.
- Human modifiable configuration – improving the odds of fixing unforeseen problems and allowing for “offline” configuration.
- Loose coupling – Avoid complex software interdependencies.
- Self contained, reproducible, builds – Provide a way for engineers to build in isolated environments, support multiple versions, etc..
- Verifiability – Choose technologies that easily support unit and regression testing.
- Flexibility – Software evolves. Admit and do it well.

These guiding principals have produced an extremely flexible object oriented software framework, primarily written in Python, for the integration of diverse building systems. Existing software modules may be inserted, removed, re-configured, inspected, re-coded and re-inserted “on the fly”. Extending the framework to support additional features is simply a matter of inheriting from the appropriate base software class, adding the required functionality to meet any unique requirements and modifying an XML based configuration file so that the new code is automatically started and linked to the rest of the framework on start-up.

When the underlying protocol (ie. BACnet) supports it; the framework provides a type of object called an “AutoDiscovered” node. In these instances, simply binding a protocol to a physical interface and starting that interface causes the framework to automatically discover and create a representation of all the devices and device attributes found on that network.
The primary role of the framework is to provide a common abstraction that can be used to represent various protocols (interfaces) along with a group of services that can then act on this generalized data. Myriads of devices are found in buildings - and each one of those devices can represent something as simple as a temperature reading in a number of different formats. Additionally, the methods for retrieving or changing these values vary by protocol.

The MPX provides a common representation of this information along with methods, ie. get() and set(), for manipulating it. Hidden behind these easy to understand abstractions, the Mediator employs sophisticated caches and batching algorithms that intelligently interact with various BAS protocols.

Once the data from these interfaces has been normalized, any number of services can be applied to it. Services typically involve creating some sort of policy that determines what to do with the data when an event occurs, and how to present this information to users or other machines. An example of a service might be, “Turn off a bank of lighting and send an e-mail to user ‘X’ when KW exceeds ‘Y’.

Essentially any information found in the framework can be accessed via the Web. “Server side” scripting is done using PSP (Python Server Pages), though most external access to the framework takes place via XML-RPC. A collection of web-based tools are provided with the Mediator that simplifies common tasks such as building web-pages and web-sites, modifying schedules and viewing real-time data. Historically, web-pages using JavaScript to update real-time data from the Mediator were the primary users of the XML-RPC interface – but we have noted an emerging trend where third parties want to develop custom applications that work with building systems – without the “hassle” of supporting numerous protocols, and the XML-RPC interface meets these needs.

4. Framework Structure

The Mediator Framework is an architecture that allows for the assembling of collaborating objects (nodes) that are then used to satisfy the functionality of a particular domain. It is designed to facilitate the rapid development and integration of nodes that can then be used to satisfy new functionality in a manner that is not tightly coupled to existing software components. It is designed to do so in a dynamic, introspective manner.
An external entity is any real world system that the Mediator represents as a collection of interacting nodes. No architecture is complete without some sort of naming system and within the Mediator framework; nodes are uniquely identified by a chain of correlating nodes that establishes a nodes relationship to its parents, grandparents, etc.. Functionality is layered so that ancestors provide specific resources and support for a node and its siblings. In the specific case of modeling a system that is used to sense and control the physical world, the URI string that uniquely identifies the node and represents the various components and functionality of that device, will be a child of the '/interfaces' node. “/interfaces” are then followed by the physical interface (com1 – com6, eth0, eth1), then protocol, using the pattern 'Protocol-Name and Protocol-Perspective (ie. Modbus Master) and one or more of the following 'types' of nodes:

- A node that represents the physical device in total.

  This is a node that would be the 'anchor' for the entire device. The pattern “Vendor-Name Model Device-Type” is typically used when providing the default name of a device. For example:

  “Trane Tracer Summit BCU”
  “Veris H8306 Power Meter”
  “Generac DG50 Generator”
  “Advantech ADAM-4011 Remote I/O”

- Organizational nodes that represent logical functional collections.

- Organizational nodes that represent physical collections.

- Nodes representing physical points.

  This is a node that represents the physical measurement by the device. If the connection point on the device is labeled, then the node will be named exactly as labeled. If the device is interpreting the value into a unitary measure, then the name will append “ (Unit-Name) “ to the point. Otherwise, according to the design of the Mediator framework, the value returned will not be interpreted. In other words, if the point is an Analog Input and the device returns an integer in the 0-1023 range for measurements on the Analog Input, then the point will be named exactly as the physically labeled point and return the value 0-1023.

- Nodes representing logical (aka virtual) points.

- Nodes representing logical conversions.

Frequently, the value returned by a point will require some sort of conversion to be useful. In these cases, an inherent child (or children) will exist for the specific purpose of converting a value. If the translated value is unitary, the name of the node will be an abbreviation of the unit. An example of this is the Dallas 18B20 temperature sensor, which returns the temperature as 1/12ths of a degree Celsius. In these cases, the point itself will not perform the conversion,
rather a child node will. This is a fundamental tenant of the framework design ... no information is hidden. Examples of complete Node Paths include:

/interfenes/eth0/BACnetIP_Client/network_1/G1P4MN1/Velocit_y_Pressure
/interfenes/eth0/BACnetIP_Client/network_1/G1P4MN1/temperature
/interfenes/eth0/BACnetIP_Client/network_1/G1P4MN1/Static_Pressure
/interfenes/com5/modbus_Client/veris8036_1/kW
/interfenes/com5/modbus_Client/veris8036_1/Volts_A-B
/interfenes/com5/modbus_Client/veris8036_1/Volts_B-C
/interfenes/com5/modbus_Client/veris8036_1/Amps_C

Ultimately a Node Tree is an acyclic directed graph of Nodes addressable within the Node namespace. A list of all running nodes may be gathered programmatically by traversing the Node Tree, using the children_names() methods. Additional details on this and other useful methods are provided in section 6 of this document. This method returns a list of the names of all children of a particular node. For example, from the '/' node, the children_names() method returns:

- aliases
- interfaces
- services

Calling the children_names() method on the '/interfaces' node, returns:

- DI1
- DI2
- DI3
- DI4
- com1
- com2
- com3
- com4
- com5
- com6
- console
- counter1
- counter2
- counter3
- counter4
- dallas1
- dallas2
- dallas3
- dallas4
- eth0
- eth1
- gpio1
• modem
• relay1
• relay2
• usb
• virtuals

This process allows all attributes of external entities represented by a running framework to be ascertained at runtime. Once a framework has been interrogated, and the details of its construction known, nodes may then be monitored and manipulated as appropriate for a given application. Typically a get() method will be called on “Inputs”, which returns the current value for a particular node or a get() or set() method will be called on “Outputs”, allowing their present state to be ascertained or overridden.

There are several areas of functionality that while technically possible using these standard methods, does not necessarily scale well. A service known as the “Subscription Manager” has therefore been designed to address several of these issues in a consistent, extensible manner, while maintaining the simplicity and elegance of get()’ing or set()’ing the underlying components. The specific areas that are addressed by the Subscription Manager are:

• Driving one node with the value of another.
• Linking node’s for the purpose of “proxying”.
• Batching the retrieval of a large number of node values in an efficient manner.
• Notifying consumers of changes to those values.
• Providing a non-blocking mechanism for consumers to read values.

Components that require the Subscription Manager locate and interact with it using a well known URI: “/services/Subscription Manager”.

A subscription can be defined as a collection of node references from which the client wants values. A client can first register a group of nodes that it is interested in and then poll a non-blocking method for a list of changed values. Each node reference in a subscription is associated with a client specific node identifier (NID) that uniquely identifies a node in a subscription. This NID provides an optimized reference to a nodes value in a subscription. Examples of uses for these optimized references include, Python keys to dictionaries, array indexes, tags used in HTML, etc..

The concepts presented this in section, though by no means all inclusive with respects to capabilities, should provide a sufficient foundation for understanding how to work with the Mediator. The fundamental building block within the framework is the node. All nodes within a running system may be discovered programmatically and then individually sent messages. A service known as the Subscription Manager exists to efficiently manage access to groups of
nodes via a subscription. All of the methods described in this section may be called via XML-RPC.

5. XML-RPC Overview

XML-RPC, which relies on HTTP for transport, was designed with the goal in mind of allowing disparate computing environments to participate in distributed computing. What this means is that an application written in Java running on a Sun Workstation, can easily and efficiently communicate with an application written in Python running on an Intel based pc or even an embedded device such as an “Internet-Enabled” PDA. XML-RPC provides the mechanisms that are necessary to make function-calls across networks – and it is well supported by a number of different programming languages, that typically employ relatively simple API’s.

An XML-RPC call takes place between two parties (client - calling process and server - called process) though in actuality an application may serve as both a client and a server, facilitating peer-to-peer communication. The below illustration demonstrates the XML-RPC communication flow between two applications over a network.

(1) The client program makes a procedure call using the appropriate XML-RPC libraries on their platform, specifying a method name, parameters, and a target server. The XML-RPC client library then takes the method names and parameters and converts them to XML.

2) The client then issues an HTTP Post request to the server with a payload that contains the XML encoded information.

(3) An HTTP server on the target machine receives the POST request and passes the message to an XML-RPC handler.

(4) The XML-RPC handler parses the XML to get the method name and parameters and then invokes the appropriate method passing it the necessary arguments.

(5) The method runs and returns a response to the XML-RPC process, which in turn encodes the response in XML.

(6) The web-server sends an HTTP Post message back to the client with XML-RPC response as the payload.
The XML-RPC client parses the XML to extract the return value – at which point it continues its normal processing.

XML-RPC calls closely model function calls in traditional programming languages, the following primitive data types are supported and can be used as call parameters or received as return values.

- **Integers**: 32bit signed integer between -2,147,483,648 and 2,147,483,647
- **Double** (floating point): Typically represented with 64bits in the range: \(-10^{-323.3}\) to \(10^{308.3}\)
- **Boolean**: 1 for true, 0 for false
- **Strings**: Any XML character is permissible within a string ... characters special to XML such as & and < are respectively represented as &amp; and &lt;.
- **Date-times**: A dateTime.iso8601 element type.
- **Binary**: ASCII-based encoding of binary objects - uses base-64 encoding.
- **Nil**: A discriminated Null value; a commonly implemented XML-RPC extension.

These simple types in turn, can be composed into more complex data structures, such as arrays or structures.

A basic example (source http://www.xmlrpcl.com) of what an XML-RPC request and response message might look like on the wire is included below:

**Request example**

```
POST /RPC2 HTTP/1.0
User-Agent: Frontier/5.1.2 (WinNT)
Host: betty.userland.com
Content-Type: text/xml
Content-length: 181

<?xml version="1.0"?>
<methodCall>
  <methodName>examples.getStateName</methodName>
  <params>
    <param>
      <value><i4>41</i4></value>
    </param>
  </params>
</methodCall>
```

**Response example**

```
```
6. Mediator XML-RPC Method Details

Within the framework an XML-RPC service exists that allows external callers to interact with the Mediator. This mechanism, referred to as rna_xmlrpc2, deploys methods that allow a caller to manage session ID’s along with a generic invoke function that allows for the executing of arbitrary public methods on any node that exists within the framework. The invoke method is called with following parameters: session_id, node_uri, method_name, method_parameters. All XML-RPC requests should be submitted to the following URL, where “_my_mediator” should be replaced with either the name or IP address of your Mediator:

http://_my_mediator/xmlrpc/

Method Name: rna_xmlrpc2.create_session:

Purpose:
Creates a unique session ID based on user\password combination that subsequent XML-RPC calls should reference. Within the Mediator a “session manager” exists that keeps track of outstanding session ID’s, expiring them after a (default) period of 60 minutes of inactivity unless the destroy_session() method is called. Session ID’s are persistent across Mediator reboots or framework restarts.

Parameters:
Username
Returns:
Session ID.

Details:
The explicit passing of user/password as parameters to the create_session() function is something we encourage for testing only. For production code, the user/password should not be passed, and will be retrieved from the Authorization header of the HTTP request associated with the XML-RPC call instead.

Rqst:

```xml
<?xml version='1.0'?>
<methodCall>
  <methodName>rna_xmlrpc2.create_session</methodName>
  <params>
    <param>
      <value><string>mpxadmin</string></value>
    </param>
    <param>
      <value><string>mpxadmin</string></value>
    </param>
  </params>
</methodCall>
```

Resp:

```xml
<?xml version='1.0'?>
<methodResponse>
  <params>
    <param>
      <value><string>BPK1ICAtABn59B5B26mt</string></value>
    </param>
  </params>
</methodResponse>
```

Method Name: rna_xmlrpc2.destroy_session:

Purpose:
Retires a session id.

Parameters:
Session ID

Details:
None
Method Name: rna_xmlrpc2.invoke: [get]

**Purpose:**
When passed ‘get’ as a third parameter, the invoke method returns the present value of a node in the framework.

**Parameters:**
- Session ID
- Node Path (ie. /interfaces/relay1)
- ‘get’

**Returns:**
The present value of a node.

**Details:**

**Rqst:**
<?xml version='1.0'?><methodCall>
  <methodName>rna_xmlrpc2.invoke</methodName>
  <params>
    <param>
      <value><string>BPK1ICAtABn59B5B26mt</string></value>
    </param>
    <param>
      <value><string>/interfaces/relay1</string></value>
    </param>
    <param>
      <value><string>get</string></value>
    </param>
  </params>
</methodCall>

**Resp:**
<?xml version='1.0'?><methodResponse>
  <params>
    <param>
      <value><int>1</int></value>
    </param>
  </params>
</methodResponse>

Method Name: rna_xmlrpc2.invoke: [set]

**Purpose:**
When passed ‘set’ as a third parameter, the invoke method sets the present value of a node in the framework to equal that of the fourth parameter.

**Parameters:**
- Session ID
- Node Path (ie. /interfaces/relay1)
- ‘set’
- Value to set the node to.

**Returns:**
Nothing is returned.

**Details:**

**Rqst:**
```xml
<?xml version='1.0'?>
<methodCall>
  <methodName>rna_xmlrpc2.invoke</methodName>
  <params>
    <param>
      <value><string>BPK1ICAtABn59B5B26mt</string></value>
    </param>
    <param>
      <value><string>/interfaces/relay1</string></value>
    </param>
    <param>
      <value><string>set</string></value>
    </param>
    <param>
      <value><int>1</int></value>
    </param>
  </params>
</methodCall>
```

**Resp:**
```xml
<?xml version='1.0'?>
<methodResponse>
  <params>
    <param>
      <value><nil/></value>
    </param>
  </params>
</methodResponse>
```

**Method Name:** rna_xmlrpc2.invoke: [children_names]

**Purpose:**
When passed ‘children_names’ as a third parameter, the invoke method returns a list of strings that represents the names of the children of the node referenced in the second parameter.

**Parameters:**
- Session ID
- Node Path (ie. /interfaces/relay1)
- ‘children_names’

**Returns:**
A list of strings containing the names of all children nodes.

**Details:**

**Rqst:**

```xml
<?xml version='1.0'?><methodCall>
<methodName>rna_xmlrpc2.invoke</methodName>
<params>
<param>
  <value><string>BPK1ICAtABn59B5B26mt</string></value>
</param>
<param>
  <value><string> /</string></value>
</param>
<param>
  <value><string>children_names</string></value>
</param>
</params>
</methodCall>
```

**Resp:**

```xml
<?xml version='1.0'?><methodResponse>
<params>
<param>
  <value><array><data>
    <value><string>aliases</string></value>
    <value><string>interfaces</string></value>
    <value><string>services</string></value>
  </data></array></value>
</param>
</params>
</methodResponse>
```

**Method Name:** rna_xmlrpc2.invoke: [create_polled]

**Purpose:**
When passed ‘create_polled’ as a third parameter, the invoke method creates a new subscription where the caller intends to poll for the values. An actual poll is a light weight method, allowing for a high poll frequency.

**Parameters:**
- Session ID
- ‘/services/Subscription Manager’
- ‘create_polled’
- A (python) dictionary of node references keyed by user supplied Node ID (NID)
- <optional> A TIMEOUT specified in seconds. If the subscription is not polled within TIMEOUT seconds, then the subscription will be destroyed.

**Returns:**
A string (SID – Subscription Identifier) that uniquely identifies the subscription.

**Details:**

**Rqst:**

```xml
<?xml version='1.0'?><methodCall><methodName>rna_xmlrpc2.invoke</methodName><params><param><value><string>BPK1ICAtABn59B5B26mt</string></value></param><param><value><string>/services/Subscription Manager</string></value></param><param><value><string>create_polled</string></value></param><param><value><struct><member><name>1</name><value><string>/interfaces/relay1</string></value></member><member><name>3</name><value><string>/services/time/local/seconds</string></value></member><member><name>2</name><value><string>/interfaces/relay2</string></value></member><member><name>4</name><value><string>/services/time/local/milliseconds</string></value></member></struct></value></param></params></methodCall>
```
Method Name: rna_xmlrpc2.invoke: [poll_all]

Purpose:
Poll the subscription service for the details of all subscribed Nodes referenced by that SID.

Parameters:
- Session ID
- ‘/services/Subscription Manager’
- ‘poll_all’
- Subscription Identifier

Returns:
A (python) dictionary, keyed by the Node ID of the Node’s “result dictionary”. The “result dictionary” has four keys: “value”, “timestamp”, “changes” and “cached”. The “value” key returns the actual value read from the Node, or the exception that prevented reading a value. The “timestamp” key returns the best estimate as to when the value was read, as a float of seconds since the epoch. The “changes” key returns the number of times the value has changed since the last poll. “Cached” is a Boolean that is false if it is guaranteed that the value was not returned from a cache. If the subscription service has not yet received an initial value for the node, then None is returned for that Node instead of a “result dictionary”.

Details:

Rqst:
<?xml version='1.0'?><methodCall>
<methodName>rna_xmlrpc2.invoke</methodName>
</methodCall>
<params>
  <param>
    <value><string>BPK1ICAtABn59B5B26mt</string></value>
  </param>
  <param>
    <value><string>/services/Subscription Manager</string></value>
  </param>
  <param>
    <value><string>poll_all</string></value>
  </param>
  <param>
    <value><string>eb40cdc7-78af-11dc-8e03-0080e4000b0d</string></value>
  </param>
</params>

Resp:

<?xml version='1.0'?>
<methodResponse>
  <params>
    <param>
      <value>
        <struct>
          <member>
            <name>1</name>
            <value>
              <struct>
                <member>
                  <name>cached</name>
                  <value><int>1</int></value>
                </member>
                <member>
                  <name>timestamp</name>
                  <value><double>1150684987.889884</double></value>
                </member>
                <member>
                  <name>changes</name>
                  <value><int>1</int></value>
                </member>
                <member>
                  <name>value</name>
                  <value><int>1</int></value>
                </member>
              </struct>
            </value>
          </member>
          <member>
            <name>3</name>
            <value>
              <struct>
                <member>
                  <name>cached</name>
                  <value><int>0</int></value>
                </member>
                <member>
                  <name>timestamp</name>
                  <value>
                    <struct>
                      <member>
                        <name>cached</name>
                        <value><int>0</int></value>
                      </member>
                      <member>
                        <name>timestamp</name>
                      </member>
                    </struct>
                  </value>
                </member>
              </struct>
            </value>
          </member>
        </struct>
      </value>
    </param>
  </params>
</methodResponse>
<value><double>1150684983.3300419</double></value>
</member>
<member>
  <name>changes</name>
  <value><int>1</int></value>
</member>
<member>
  <name>value</name>
  <value><string>/services/time/local/seconds</string></value>
</member>
</struct>
</value>
</member>
<member>
  <name>2</name>
  <value>
    <struct>
      <member>
        <name>cached</name>
        <value><int>1</int></value>
      </member>
      <member>
        <name>timestamp</name>
        <value><double>1150684987.8970649</double></value>
      </member>
      <member>
        <name>changes</name>
        <value><int>1</int></value>
      </member>
      <member>
        <name>value</name>
        <value><int>0</int></value>
      </member>
    </struct>
  </value>
</member>
<member>
  <name>4</name>
  <value>
    <struct>
      <member>
        <name>cached</name>
        <value><int>1</int></value>
      </member>
      <member>
        <name>timestamp</name>
        <value><double>1150684987.8970649</double></value>
      </member>
      <member>
        <name>changes</name>
        <value><int>2</int></value>
      </member>
      <member>
        <name>value</name>
        <value><double>902.75907516479492</double></value>
      </member>
    </struct>
  </value>
</member>
Method Name: rna_xmlrpc2.invoke(): [poll_changed()]

**Purpose:**
Poll the subscription service for all subscribed Node’s whose values have changed since the last time poll_changed() was invoked or since the subscription was created, which ever is more recent.

**Parameters:**
- Session ID
- Node Path (ie. /interfaces/relay1)
- ‘poll_changed’

**Returns:**
A dictionary, keyed by the Node ID of the changed values "result dictionary". The "result dictionary" dictionary has three keys: "value", "timestamp" and "cached". The "value" key returns the actual value read from the Node, or the exception that prevented reading a value. The "timestamp" key returns that best estimate as to when the value was read, as a float of seconds since the epoch. The “changes” key returns the number of times the value has changed since the last poll. "Cached" is a boolean that is false if it is guaranteed that the value was not returned from a cache.

**Details:**

**Rqst:**

```xml
<?xml version='1.0'?>
<methodCall>
  <methodName>rna_xmlrpc2.invoke</methodName>
  <params>
    <param>
      <value>
        <string>BPK1ICAtABn59B5B26mt</string>
      </value>
    </param>
    <param>
      <value>
        <string>BPK1ICAtABn59B5B26mr</string>
      </value>
    </param>
    <param>
      <value>
        <string>BPK1ICAtABn59B5B26mt</string>
      </value>
    </param>
  </params>
</methodCall>
```
<methodCall>
  <params>
    <param>
      <value>
        <string>/services/Subscription Manager</string>
      </value>
    </param>
    <param>
      <value>
        <string>poll_changed</string>
      </value>
    </param>
    <param>
      <value>
        <string>eb40cdc7-78af-11dc-8e03-0080e4000b0d</string>
      </value>
    </param>
  </params>
</methodCall>

Resp:

<?xml version='1.0'?>
<methodResponse>
  <params>
    <param>
      <value>
        <struct>
          <member>
            <name>3</name>
            <value>
              <struct>
                <member>
                  <name>cached</name>
                  <value><int>0</int></value>
                </member>
                <member>
                  <name>timestamp</name>
                  <value><double>1150684983.3300419</double></value>
                </member>
                <member>
                  <name>changes</name>
                  <value><int>1</int></value>
                </member>
                <member>
                  <name>value</name>
                  <value><string>/services/time/local/seconds</string></value>
                </member>
              </struct>
            </value>
          </member>
        </struct>
      </value>
    </param>
    <param>
      <value>
        <struct>
          <member>
            <name>cached</name>
            <value><int>1</int></value>
          </member>
        </struct>
      </value>
    </param>
  </params>
</methodResponse>
<member>
  <name>timestamp</name>
  <value><double>1150684987.8970649</double></value>
</member>

<member>
  <name>changes</name>
  <value><int>2</int></value>
</member>

<member>
  <name>value</name>
  <value><double>902.75907516479492</double></value>
</member>
</struct>
</value>
</member>
</struct>
</value>
</param>
</params>
</methodResponse>