

# **Reality**

## **A Platform for Distributed SCADA**

**Developer's Guide  
(Preliminary Information)**

---

## **Reality: Developer's Guide (preliminary information)**

Version 0.4.3

Published 2007-05-08

Copyright © 2006-2007 NC Engineering Ltd.

---

## Table of Contents

Introduction.....	1
System Architecture.....	1
Design Principles.....	2
Security Model.....	2
Core Concepts.....	3
Reality Server.....	3
Producers.....	3
Consumers.....	3
Nodes.....	3
States.....	3
Sessions.....	4
Permissions.....	4
The Application Programming Interface (API).....	5
Making Requests.....	5
Request Parameters.....	5
Result codes.....	6
Recovering from Errors.....	6
XML Format.....	7
JSON Format.....	9
Node References.....	9
Specifiers.....	10
Timestamp Formats.....	13
Time Ranges.....	13
Operational Scenarios.....	14
Consuming Real-Time Data Updates.....	14
Producing Real-Time Data.....	16
Passing Messages Between Machines.....	18
Embedding Reality Data in Web Pages.....	19
Query Reference.....	20
state (GET).....	20
history (GET).....	21
set (POST).....	22
write (POST).....	22
open (POST).....	23
close (POST).....	24
events (GET).....	25
touch (POST).....	26
signal (POST).....	26
sub (POST).....	26
unsub (POST).....	27
exec (POST).....	28



---

## Introduction

Reality is a platform for distributed SCADA applications. Reality aims to solve many of the problems associated with distributed SCADA applications, such as performance, network configuration and security.

With Reality it is possible to integrate a variety of platforms in disparate locations, including mobile devices, using a unique architecture which allows machines to communicate securely without requiring additional network configuration, such as VPN's.

In Reality, data items are first-class citizens, each variable having its own URI, or universal resource identifier. URI's make it easy to track, analyze, extract, and otherwise manipulate real-time and historical data. Reality generates historical reports on the fly, in a variety of text and graphic formats.

Reality lets clients subscribe to specific nodes and receive change events as soon as changes occur. The system also makes it easy for data producers such as PLC's and other remote terminal units (RTU's) to track live sets of variables, relieving them of the burden of managing client state.

## System Architecture

Reality makes it possible to connect together various machines to create a SCADA application by using a network topology with clearly-defined machine roles; implementing a zero-configuration hierarchical namespace for all application variables and parameters; and providing an HTTP-based protocol to synchronize application state between machines. In fact, the Reality architecture makes it possible to keep configuration of each machine down to a single parameter: the server URI, which acts as the entry point to the system.

Reality utilizes a centralized network topology, in which one or more servers are used to manage the application state and synchronize all machines in the application. A consumer is a machine which monitors application variables. A producer is a machine which is usually directly connected to a PLC, and generates the state of application variables, and also handles write requests to change those variables. A single machine can operate as both a consumer and a producer at the same time. The Reality server maintains the current application state, logs historical records, and manages subscriptions.

---

## Design Principles

Reality was designed with the following principles in mind:

- Platform independence – The system should be accessible from as many computing platforms as is possible. The API should be accessible using any development tool (i.e., not tied to a specific development platform.)
- Location transparency – Machines can be placed anywhere and be accessed from anywhere.
- Simplicity – The system should be as simple as possible to operate and manage.
- Minimum configuration effort – The users should be relieved of unnecessary configuration.
- Secure operation – The security model should provide a way to set fine-grained permissions to specific users for specific variables.
- Fault tolerance – The System should make it easy to recover from failures. It should also fail gracefully.
- Near-zero maintenance – The system should be able to keep ticking without human intervention (as much as possible.)

## Security Model

Reality includes a fine-grained access control mechanism that can be used to ensure only specific users have access to specific parts of the namespace. Users can furthermore be allowed only certain kinds of actions on specific nodes. For example, a user can be allowed to read the value of a node, but not to change it.

All clients are expected to provide user credentials when accessing the server. If no credentials are supplied, the default set of permissions is assumed. Once a client signs in with user credentials, the server creates a session for the client, which acts as a security context. The session is temporary and must be renewed from time to time by signing in again. The client can also close its session at any time. Sessions are identified by a session key (skey), a special string which is valid only for the client's IP address and expires after a set amount of time. User credentials are passed only at login, and are not stored or transferred in response bodies or cookies.

Users are identified by a node path. In fact, users are special nodes. The hierarchical arrangement of users in the namespace allows one to grant or revoke permissions to groups of users instead of having to repeat the configuration for each user, while leaving open the possibility of changing the permissions for a single user.

Reality also logs user operations to a database for security auditing.

---

## Core Concepts

### Reality Server

The Reality server is a single machine or an array of machines that forms the center of the Reality system. The server is responsible for maintaining application state, providing data to clients, and coordinating consumers and producers.

For application developers, the Reality server provides an opaque service which can be accessed with a URI using the HTTP protocol.

### Producers

A producer is a client that generates real-time data. A producer can receive events from the server that tell it what nodes are currently being monitored by other clients. The Reality server takes on the task of consolidating subscription lists from all consumers, letting the producer concentrate on supplying the real-time data.

A producer can own a part of the application namespace, so that any write requests to any node in that part is sent to the producer as an event.

A producer can also receive free-form messages from other clients to perform special operations.

### Consumers

A consumer is a client that consumes real-time and historical data. A client can be a consumer and a producer simultaneously. A consumer can subscribe to nodes and receive events whenever the a node's state changes.

### Nodes

A node is a single variable that has a single state at any given time. A node is referenced using a **node path**, much like a directory path. All nodes are related to each other and are arranged in a hierarchical tree-like structure called the **namespace**, at the top of which is the **root node**. A node can have zero or more sub-nodes. The hierarchical arrangement of nodes is significant for the inheritance of the node's configuration, such as the default sample rate or other producer specific configuration.

Nodes can be created on the fly, simply by referencing it. Nodes can also be removed, browsed, searched, configured and otherwise manipulated using the Reality API.

Nodes are identified by node paths. Node paths work similarly to directory paths. The components of a node path are delimited by forward slashes. Node paths make it easy to refer to nodes by providing a uniform and universal format.

### States

A state represents the condition of a node at a specific time. A state has the following attributes:

- **Stamp** – the time and date the condition was encountered.
- **Quality** – the kind of condition: unknown, good, bad, invalid, forced or simulated.

- 
- **Value** – the node's value.
  - **Data type** – an optional number denoting the value's data type: text, number or boolean.

Since in a distributed system the real-time value of a variable can only be estimated, Reality provides the **last known state** instead. By default, all state transitions are recorded to a historical states table, and can be recalled by using historical queries.

## Sessions

A session represents a conversation between a client and the reality server. A session provides a temporary security context for accessing the server. A client can create a new session by providing user credentials - i.e. a user name and a password. Sessions are identified by a session key (**skey** for short), and are valid only for a specific IP address.

A sessions will automatically expire if it is older than 3 hours or haven't been used for 30 minutes. If a client discovers its sessions has expired, it can simply create a new one by signing in again.

A session also provides a context for event-based messaging between machines. Clients can use sessions to subscribe to specific nodes, send messages to other clients, or update node states.

Sessions can be mounted at a specific node in order to create an end-point that can receive messages from other clients. This technique is also used by data producers to receive consolidated subscription lists from the server. Once a session has been mounted at a node, the client is considered the owner of the node and all of its sub-nodes. The server generates events to keep the client updated with the list of items required by other clients at any given moment.

## Permissions

Permissions can be granted or revoked for certain users to perform specific operations on specific nodes. The system currently supports three kinds of permissions: read, write and configure.

When a client issues a request to access a specific node, the Reality server calculates permissions for the client's user account and the specified node. If the client is not signed in, the anonymous user account is used.

Permissions are automatically inherited so they can be defined only once per user. For example, if a user was granted permission to read from /cex, the user would by default have permission to read from /cex/40130 or any other subnode of /cex.

Another form of permission inheritance is when setting permissions for groups of users. Since user accounts are nodes, they can be arranged hierarchically like all other nodes. For example, if the system administrator grants configure rights on /cex to the /users user account, then /users/david, or any user inside /users would be granted configure rights on /cex as well.



---

# The Application Programming Interface (API)

## Making Requests

The Reality API is based on HTTP. Rather than exposing an object model, the API strives to follow a functional design, and make it easy to get at the needed data.

Each request is made of the following elements:

- **HTTP method** – in the Reality API we only use GET and POST. The GET method is used for querying (“reading”) the server, and the POST method is used for changing (“writing”) data on the server.
- **URL Path** – the path of the node to be accessed. As you shall see, if multiple nodes are to be accessed in a single request, the node path signifies the base path for resolving relative paths specified elsewhere in the request.
- **Parameters** – an optional set of parameters to further qualify the request.

The response returned for a request depends on the parameters passed in the request. The HTTP protocol allows the transfer of a variety of data formats in the response body, and the Reality API takes advantage of this to a great extent. The client can specify the desired data format to be returned in the response.

Reality allows parameters to be passed in a variety of ways. By default parameters are encoded as part of the URL. This is especially beneficial for GET operations because it allows a user to bookmark highly nuanced URLs that can be used to receive complex responses. Parameters can also be passed in the request body. This is the default for POST operations. By default parameters are URL-encoded, but can also be specified in XML format.

In order for the Reality server to process parameters in the request body, the client should include a Content-Type header specifying the format used. For URL-encoded parameters, the format should be `application/x-www-form-urlencoded`. For XML, the format should be `text-xml`. The XML format for request parameters is discussed in detail on page 9.

## Request Parameters

The API specifies a number of parameters that can be used in most of the different types of requests in order further qualify the request. In addition, each request type can use additional parameters for further specialization. Some parameters are optional and can be omitted from the request. If a parameter is omitted, a default value is assumed.

The following parameters provide the basic semantics for Reality API requests:

Name	Default value	Description
query/q	node_state for GET requests, node_write for POST requests.	The type of query to perform.
flavor/f	usually xml, but depends on the specific query.	The data format for rendering the response.
skey	n/a	The session key obtained by the client upon successful login.

---

Name	Default value	Description
nodes/ node/n	n/a	One or more node paths on which to perform the query, separated by an exclamation mark (!). The paths specified can be either relative or absolute. If this parameter is omitted, the path of the request is used as the target node path.
mode/m	normal	The synchronization mode to use. Some requests support blocking on the server until a condition is fulfilled or until a timeout period has elapsed.

The query, flavor, nodes and mode parameters can be specified by their shorthand notation, q, f, n and m respectively.

## Result codes

The API defines a standard set of result codes. The result code signifies whether the request has been executed successfully, or whether a specific error has occurred. Result codes are included in XML and JSON requests.

If an error occurs, the response will include the result code and a message describing the error. If a flavor other than XML or JSON is requested or implied, an HTML page is rendered with the error information.

The following result codes are defined:

Code	Description
OK	The request was executed successfully.
InvalidHTTPMethodError	An invalid method was used to perform the specified query. Each query type specifies the HTTP method to use (GET or POST.)
InvalidNodeError	An invalid node was referenced.
InvalidNodeDepthError	An invalid node depth was specified.
InvalidQueryError	An invalid query was specified.
InvalidFlavorError	An invalid flavor was specified. Please note that each query type supports a different set of flavors.
InvalidQueryParameterError	An invalid parameter value was specified.
MissingQueryParameterError	A mandatory parameter was not included in the request.
InvalidCredentialsError	Invalid user credentials were supplied (user name and/or password.)
AccessDeniedError	The client does not have permission to perform the request.
InvalidSessionError	An invalid or expired session key was specified.

## Recovering from Errors

The handling of errors can change depending on the kind of client involved. For example, an unattended client, such as an RTU, would retry the request or drop it, according to the specific type of error. An HMI client might, on the other hand, display an error message to the user.

---

There are, however, some errors which can be recovered from by following a standard procedure. These are discussed in the following paragraphs.

### **AccessDeniedError**

This result code is returned when the client is refused permission to perform the specified query on the specified node or nodes. If the client has not yet logged in, it should log in with user credentials and retry the request with the newly received session key. If the client has already logged in, it can inform the user that the request failed because permission was refused.

### **InvalidSessionError**

This result code is returned usually when the specified session has expired. As a matter of principle all sessions expire in order to increase security. When a session has expired, the client should issue a new open request with the user credentials, and retry the request using the newly received session key. If the client was subscribed to any nodes, it should renew its subscription.

## **XML Format**

The XML format used in response follows the following principles:

- Namespaces are not used.
- Tag attributes are not used.
- The character encoding used is UTF-8.
- The response is enclosed in a <reality> tag.
- A <result> tag contains the result code.
- A <message> tag contains the error message in case of an error.
- Null or empty values are omitted from the response.

A normal XML response looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<reality>
  <result>OK</result>
  <node>
    <id>12</id>
    <path>/cex/41030</path>
    <quality>1</quality>
    <stamp>Thu Jun 15 16:44:39 UTC 2006</stamp>
    <value>331</value>
    <datatype>1</datatype>
  </node>
</reality>
```

An XML error response might look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
```

---

```
<reality>
  <result>InvalidNodeError</result>
  <message>Invalid node (/ads)</message>
</reality>
```

## Specifying Request Parameters in XML Format

The Reality server can also accept request parameters in XML format instead of the normally used URL encoding. The format for specifying queries is a tag with the query specifier enclosing multiple tags for each of the query values. For example, the following query:

```
/cex/40301?q=history&t=lastmonth&f=js
```

Can be specified in XML as follows:

```
<history>
  <path>/cex/40301</path>
  <t>lastmonth</t>
  <f>js</f>
</history>
```

When parameters are specified in XML format, the application must set the Content-type HTTP header to text/xml.

## Multiple Queries in a Single Request

A client can specify multiple queries in a single request by using XML formatting for request parameters. In the current API version, this is limited to POST requests only. To execute multiple queries, the client should enclose all queries in a <execute> tag, as in the following example:

```
<execute>
  <write>
    <path>/cex/40130</path>
    <v>23</v>
    <dt>integer</dt>
  </write>
  <sub>
    <path>/cex/9843</path>
  </sub>
</execute>
```

A client can also use relative paths to reference nodes by including a context path tag inside <execute>, and using <node> or <nodes> instead of <path> for each query:

```
<execute>
  <path>/cex</path>
  <write>
    <node>40130</node>
    <v>23</v>
```

---

```
</write>
<sub>
  <node>/cex/9843</node>
</sub>
</execute>
```

## JSON Format

The JSON format is much more compact than XML, and though it lacks parsers in many languages (especially “old” ones), it is extremely beneficial to Javascript clients, since it can be *eval*ed directly without any parsing. The JSON format, like XML, allows nested structures to be defined, but unlike XML, each structure can include only one value for any 'tag'. To solve this we use arrays:

```
{
  "result": "OK",
  "nodes": [
    {"quality": 5, "datatype": 0, "value": "331",
     "stamp": 1150379079.45837, "path": "/cex/41030"},
    {"quality": 0, "datatype": 0, "value": null,
     "stamp": 1150379081.34813, "path": "/cex/41031"}
  ]
}
```

(The example above includes line breaks and indented nested elements for the sake of visual clarity. These are omitted in the actual server response.)

In the example above, separate node structures are contained in a nodes array. Contrast this to the use of several <node> tags in an XML response. Also notice that field values are formatted differently - the quality and datatype fields are expressed as integers, and the stamp field is expressed as a floating-point number.

An error response in JSON format would look like this:

```
{
  "result": "InvalidNodeError",
  "message": "Invalid node (/ads)"
}
```

## Node References

Nodes can be referenced either by path or by id. A node id is an internal number the server assigns to each node. The id is guaranteed to stay the same as long as the node exists. If the node has been deleted and recreated, its id might be different. As of this version of the API, the use of id's for reference is not recommended, though the server will always include a node's id whenever it is referenced in a response.

---

A node path is a text string made of multiple components delimited by forward slashes. Each component denotes a nested level in the application namespace. For example, the node path `/west/tank 1/level` denotes a node named 'level' nested inside a node named 'tank 1' nested inside a node named 'west' nested inside the root node. The root node is referenced by a single slash (/). The last component of a path denotes the node's name. The only node which has no name is the root node. All other nodes must have a name.

### **Absolute and Relative Paths**

Absolute node paths start with a forward slash. Relative node paths omit the first slash. Relative node paths can be used in the node/nodes parameter, as discussed above. When relative paths are specified, they are considered by the server to stem from the path of the request. For example, if the following request URI has been specified:

```
GET /cex?q=state&nodes=40130!40131
```

The server will return the node state for both `/cex/40130` and `/cex/40131`.

Node path are validated according to the following rules:

- A path component cannot contain any of the following characters: `:`, `/`, `\`, `#`, `|`, `!`, `{` and `}`. This also means that these characters cannot be used in node names.
- A path cannot contain two consecutive slashes, that is a path component cannot be empty.
- A path cannot end with a slash (except for the root path.)

## **Specifiers**

Specifiers are values that have special meaning in the Reality API. Specifiers can be used in both request parameters and responses. Some specifiers can be expressed by either a code - a short lowercase word - or a corresponding integer number. In that case, either can be used as a paramter value when performing a request.

### **Query Specifier**

Query specifiers are used to tell the server what action the client would like to perform on the referenced node. Queries are grouped in several categories. The query specifier in most cases includes the category name and the action name, separated by an underscore. Some queries, especially the frequently-used ones, also have a shorthand notation.

---

Following is a list of the currently available query types:

<b>Code</b>	<b>Description</b>
<b>Node-related Queries</b>	
state	Retrieve a node's current state.
history	Retrieve a node's history.
set	Set a node's current state.
write	Issue a write request for the node.
<b>Session-related Queries</b>	
open	Open a session.
close	Close a session.
events	Retrieve events for the session.
touch	Update time stamps for all produced nodes for the session.
sub	Subscribe to a node.
unsub	Unsubscribe to a node.
signal	Request updates for all subscribed nodes.
<b>Miscellaneous Queries</b>	
execute	Execute multiple queries at once.
help	Retrieve API documentation.

## Flavor Specifier

The flavor specifier is used to tell the server in which format to render the response. If the flavor specifier is omitted, the default flavor will vary according to the query specified. In addition, each query type supports a different set of flavors. The flavor used also affects the Content-Type HTTP header that is returned together with the response.

Currently the system supports the following flavors:

<b>Code</b>	<b>Content-Type</b>	<b>Description</b>
xml	text/xml	XML format
js	text/javascript	JSON format
html	text/html	HTML format
html_partial	text/html	HTML format (without <html> and <body> tags)
csv	text/csv	
gif	image/gif	
png	image/png	

## Quality Specifier

The quality specifier signifies the kind of condition a node is in at the time it was recorded. Qualities are specified by code number. The following qualities are defined:

---

Code	Number	Description
unknown	0	The state of the node is unknown.
good	1	The node value was sampled successfully.
bad	2	The node value could not be sampled. This code is used by a producer to denote a communication error with an RTU, or a similar condition.
invalid	3	The node is considered invalid. This code is used by a producer to denote a node reference that does not exist on the RTU.
simulated	4	The node value is simulated, i.e., is not connected with a physical phenomenon.
forced	5	The node value is forced by a client. This code is used when clients write to a node that is not currently produced.

### Datatype Specifier

Reality does not force clients to specify data types when writing or updating node values, nor does the server place any restriction on- or assumptions about data types. All Reality values are stored and transmitted as text.

However, once a datatype is attached to a value, it is always supplied together with the value. For example, if a data producer has updated the value of a node and specified a boolean data type, all consumers subscribed to that node would receive the new value together with the boolean datatype specifier. Of course, a data producer may change the datatype of a node's value at any time, together with a change of value.

Datatypes are beneficial to the system because they allow clients to render node values correctly. For example, an HMI client can present a check box for a boolean value, as opposed to an edit box for a text value.

Future versions of Reality will include functionality for automatic conversion between data types. In addition, a unit specifier could be attached to the value to signify engineering units. The system then could include rules for conversion of units, e.g., convert m<sup>3</sup> to ft<sup>3</sup>, or kg/m<sup>2</sup> to p/ft<sup>2</sup>, etc.



---

The following datatypes are currently defined:

<b>Code</b>	<b>Number</b>	<b>Description</b>
text	0	The value is a text value.
integer	1	The value is a signed integer.
float	2	The value is a floating-point number.
boolean	3	The value is a boolean value. In this case, the value itself is expected to contain either a number (where 0 is considered false and anything else true), or a textual representation of a boolean value (either 'true' or 'false'.)

Datatype specifiers can also be used by data consumers to perform write requests. In that case, the datatype specifier is passed together with the value to the data producer, and it is the responsibility of the data producer to make the translation from the specified data type to the actual datatype in the RTU.

### **Event Kind Specifier**

Event kind specifiers are used to signify different kinds of events. Events are generated by the server and can only be consumed by clients. The only way for a client to directly generate events is to send a message to another client. Events are used by the server to tell clients when the application state changes in a way that should interest the client. For example, if a client subscribes to a node, then the server will generate an event for the client whenever the node's state changes. The client can then retrieve the event by issuing a events query.

The following event kinds are defined:

<b>Code</b>	<b>Number</b>	<b>Meaning</b>
message	0	A message received from another client.
start_track	1	The data producer should start tracking the specified node.
stop_track	2	The data producer should stop tracking the specified node.
update_track	3	The data producer should update the sampling rate and other configuration for the specified node.
write_request	4	The data producer should perform a write request on the specified node.
state_changed	5	The state of a subscribed node has changed.

### **Mode Specifier**

Mode specifiers are used in a number of requests that allow long-standing, or blocking requests. Blocking requests are requests in which the server does not immediately respond, but rather keeps the connection open and eventually renders a response once a condition has been fulfilled, or a timeout period has elapsed, whichever comes first.

The events query also allows a streaming mode, in which the server sends events to the client as they occur, while keeping the connection open for an extended period of time.

---

The following modes are defined:

Code	Number	Meaning
async	0	The response is rendered immediately.
block	1	The server blocks until a condition has been fulfilled or until the specified timeout period has elapsed.
stream	2	The server streams chunks of the response according to the query behavior until the specified timeout period has elapsed.

## Timestamp Formats

The timestamps returned in responses vary in format according to the flavor requested. XML response return timestamps in RFC 2822 format:

day-of-week, DD month-name CCYY hh:mm:ss zone

where the zone is UTC (GMT).

When the flavor requested is JSON, timestamps are rendered as a floating-point number representing the number of seconds since UNIX epoch (January 1, 1970 at midnight UTC).

## Time Ranges

Time ranges specified for historical queries can be expressed in a variety of ways. The Reality server can understand a variety of partial or complete time formats, and can also understand (to a degree) human language.

A time duration can include either a single specifier or a pair of specifiers separated by a comma or the word 'to'.

The following are examples of valid time durations:

Specifier	Meaning
2004 06 to 2004-08	June 1, 2004 to August 31, 2004
Feb 1 to Feb 8	February 1 of this year to a Feb 8 (at day end) of this year
last Monday to next Sunday	Last monday to next Sunday (at day end)
last week	Last monday through Sunday (at day end)
last month	The previous month (from first to last day at day end)
last year	The previous year (from first to last day at day end)
yesterday	Yesterday from 00:00 to 24:00
past week	A week ago to now
past month	A week ago to now
this month	The current month (from first to last day at day end)
12:24 to 12:37	Today from 12:24:00 to 12:37:59
3 hours	The last 3 hours
4 days 12 hours	The last 4 days and a half
10 minutes	The last 10 minutes

Notice that ranges are always considered inclusive. Also notice that it is also possible to specify times in the future as well as in the past.

---

## Operational Scenarios

The following section discusses some possible uses of the API by clients in various scenarios. It is not comprehensive in any way, nor do clients have to function in the exact same way to achieve value from the system. The API was designed to allow many different uses.

In the examples that ensue, the requests are illustrated in a shorthand manner, that is, many of the HTTP headers are omitted. The omitted headers are replaced by an ellipsis (...). It is assumed that the reader is familiar with how HTTP requests look.

### Consuming Real-Time Data Updates

A real-time data consuming client starts interacting with the server by opening a session. A session supplies a context in which the client subscribes to arbitrary nodes and receives updates to node states. When the client is done, it closes the session.

#### Opening a Session

To open a session, the client issues a open request, supplying a user name and a password:

```
POST /  
...  
q=open&u=joe&p=16792403
```

To which the server responds:

```
<reality>  
  <result>OK</result>  
  <skey>7b39d890731523f1b37e79d9f8cf5b45</skey>  
</reality>
```

The client extracts the session key (skey) from the response and keeps it for usage in later requests. Notice that as a security measure the skey is always a 128-bit hash value.

#### Subscribing to a Node

To subscribe to a node, the client issues a sub request:

```
POST /cex/40130  
...  
q=sub&skey=7b39d890731523f1b37e79d9f8cf5b45
```

The response from the server is as follows:

```
<reality>  
  <result>OK</result>  
</reality>
```

From here on, the client periodically requests events from the server with an events query. As will be discussed in the query reference, the client can ask the server to keep the connection open and send events as they are generated.

---

## Retrieving Session Events

To retrieve events, the client issues the following request:

```
GET /?q=events&skey=7b39d890731523f1b37e79d9f8cf5b45&m=stream&t=60
```

To which the server responds with:

```
<reality>
  <event>
    <kind>state_changed</kind>
    <node_id>24</node_id>
    <node_path>/cex/40130</node_path>
    <quality>good</quality>
    <stamp>Thu Jun 15 12:32:55 UTC 2006</stamp>
    <value>13</value>
    <datatype>integer</datatype>
  </event>
  (30 seconds pass)
  <event>
    <kind>state_changed</kind>
    <node_id>24</node_id>
    <node_path>/cex/40130</node_path>
    <quality>good</quality>
    <stamp>Thu Jun 15 12:33:25 UTC 2006</stamp>
    <value>14</value>
    <datatype>integer</datatype>
  </event>
  (another 30 seconds pass)
</reality>
```

The client has specified that the server should stream events as they occur and timeout after 60 seconds. The connection stays open for the 60 seconds, during which the server checks for any events for the specified session, and adds an event tag to the response for each occurrence.

In the above response, the server generated two events. When a client first subscribes to a node, the server will generate a 'first update' event just to get the client up to . When the node's value changes, the client will have received another event.

This is really all there is to it! When the client no longer needs updates for a node, it issues a unsub request, in similar fashion to sub. When the client is done conversing with the server, it issues a close request. Note that this is not mandatory. The server automatically cleans up after clients if they leave sessions and subscriptions about.

## Writing Values

In order to change real-time values, the client can issue a write request:

```
POST /cex/40130
```

---

```
...
q=write&value=46
```

To which the server responds:

```
<reality>
  <result>OK</result>
</reality>
```

The client can also ask the server to block until the value is updated. In that case, a timeout must be specified, since there is no guarantee that the operation would complete in any amount of time. If the node's value has not changed before the timeout period has elapsed, the server returns an error response.

```
POST /cex/40130
...
q=write&value=46&m=block&t=10
```

The server responds with an event:

```
<reality>
  <result>OK</result>
  <event>
    <kind>state_changed</kind>
    <node_id>24</node_id>
    <node_path>/cex/40130</node_path>
    <quality>good</quality>
    <stamp>Thu Jun 15 12:34:03 UTC 2006</stamp>
    <value>46</value>
    <datatype>integer</datatype>
  </event>
</reality>
```

In the above example, the client asked the server to block until the value has changed for a maximum of 10 seconds. The server responded with an event representing the new state of the node.

## Producing Real-Time Data

Data producers have two ways of updating real-time data on the Reality server. The first, and simpler way is to simply use `node_set` requests to update node values. Notice that `node_set` queries have different effects than `node_write` queries. This is discussed in more detail in the query reference. Suffice it to say that `node_write` is used by consumers and `node_set` by producers.

The second manner of producing data involves opening a session and mounting it at a node. This allows a data producer to get notified which nodes are being subscribed to at any time. The Reality server consolidates the subscription requests from all consumers and can supply to a producer an up-to-date list of nodes to track. The server also supplies along with each node reference a preferred sample rate for the node.

---

This feature of the Reality server allows the creation of sophisticated data producers with very little effort. Since the data producer does not have to deal with talking to multiple clients and managing subscriptions, it can simply deal with supplying the data to the server.

---

## Mounting a Session

Like a data consumer, a producer starts by opening a session:

```
POST /plc1
...
q=open&mount=yes&u=plc1&p=plc1
```

In the above request, the client has specified that it would like to mount the session at the node path `/plc1`. The act of mounting a session means that the client now owns `/plc1` and all of its sub-nodes.

From here on, the client is responsible for providing values for `/plc1` and all its sub-nodes, and will receive events telling it which nodes need to be tracked.

Upon issuing an events request, the server responds with the following hypothetical events:

```
<event>
  <kind>start_track</kind>
  <node_id>13</node_id>
  <node_path>V3</node_path>
  <sample_rate>10</sample_rate>
</event>
...
<event>
  <kind>stop_track</kind>
  <node_id>53</node_id>
  <node_path>V39</node_path>
</event>
```

This tells the client to start tracking `/plc1/V3` with a sample rate of 10 seconds. The sample rate is considered the preferred sample rate. The producer is not required to enforce this sample rate – it is only used to give a clear indication of whether a node's state is stale or fresh. If multiple consumers subscribing to the same node specify different sample rates, the server will use the smallest rate as best rate. If the best rate changes, the server will notify the producer with an `update_track` event.

The client is also told to stop tracking `/plc1/V39`. Notice the use of relative node paths in the response.

## Updating Node Values

The data producer is expected to perform a 'first update' for each newly tracked node in order to bring the server up to date, and then update it whenever a value changes. When a new value has been acquired, the producer issues a `node_set` request:

```
POST /plc1
...
<execute>
  <node_set>
    <node>V3</node>
    <quality>good</quality>
```

---

```
<value>5428</value>
<datatype>integer</datatype>
</node_set>
<node_set>
  <node>431</node>
  <quality>invalid</quality>
</node_set>
<node_set>
  <node>V9</node>
  <quality>bad</quality>
</node_set>
</execute>
```

To which the server responds:

```
<reality>
  <result>OK</result>
</reality>
```

Since '341' is not recognized by the producer as a valid address, it updated /plc1/341 with 'invalid' quality. This change of state is communicated to all clients interested in this node, and lets them understand the node is not valid without having to be aware in advance of the producer's address space. The producer has also updated /plc1/V9 with a 'bad' quality. This means that the producer could not have acquired the value of V9, although it is a valid address.

## Updating Timestamps

The fact that a producer is only required to update a node's state when its value changes, leads to a situation where if a node's value stays the same for a long time, the timestamp associated with the node's state will be considered old. It is possible to periodically update all tracked nodes irrespective of whether their values changed. This might, however, lead to very high bandwidth and is somewhat inefficient. In order to help update timestamps, the Reality API includes the touch query:

```
POST /
...
q=touch&skey=7b39d890731523f1b37e79d9f8cf5b45
```

The effect of this operation is that all timestamps for currently tracked nodes are brought as close as possible to the current time, by advancing them according to the specified sample rate for each node. For example, if the current time is 18:56:23, and we have a node who's latest timestamp is 18:56:04 and which is tracked with a sample rate of 10 seconds, the server will advance its timestamp to 18:56:14.

When timestamps are advanced, the new state is not recorded in history, but all subscribed clients are informed with the updated node state.



---

## Passing Messages Between Machines

Clients connected to the same server can send messages to each other. This is useful in a variety of scenarios, for example using Reality as a gateway through a programming workstation can talk to a remote PLC:

```
/POST /cex
...
<message>
  <source>/workstation</source>
  <body>:110400080001</body>
  <mode>block</mode>
  <timeout>10</timeout>
</message>
```

The server responds as follows:

```
<reality>
  <result>OK</result>
  <event>
    <kind>message</kind>
    <source>/cex</source>
    <body>:110402000A</body>
  </event>
</reality>
```

In the above example, the client has sent a ModBus message to /cex, and has asked the server to block until a message comes back, for a maximum of 10 seconds. The server responded with an event containing the reply from /cex.

Notice that the target of a message is identified by a path. In order to receive messages, a client must open a session and mount itself at a node path. If the sender of a message would like to receive a reply, it must also be mounted.

## Embedding Reality Data in Web Pages

One of the most profound benefits of using an HTTP-based API is the ability to embed data provided by the server in regular web pages, with almost no effort. Consider the following HTML document:

```
<html>
  <body>
    <h2>A table of current data from Reality</h2>
    <iframe src="http://reality-scada.com/cex/?n=40130!40132!40133
    &q=node_state&f=html_partial"/>
  </body>
</html>
```

The resulting web page would include a frame containing a table with the current values of /cex/40130, /cex/40132 and /cex/40133.

One can also embed images:

---

```
<html>
  <body>
    <h2>Last 24 hours of /cex/40130</h2>
    
  </body>
</html>
```

---

## Query Reference

The following reference lists all the currently supported query types, each with a list of parameters, response formats and possible error result codes. The query type is followed by the appropriate HTTP method in parentheses.

### state (GET)

The state query is used to retrieve the last known state of one or more nodes.

#### Mandatory Parameters

URI path - the node path, or a base path (used in conjunction with the nodes parameter.)

#### Optional Parameters

node/nodes/n - one or more relative or absolute node paths separated by an exclamation mark. If this parameter is included in the request, the URI path is considered as a base path for extending relative paths.

#### Access Control

The client must have read permission for the specified nodes.

#### XML Response (Default Flavor)

For each referenced node, a <node> tag is included in the response, containing the following tags: id, path, quality, stamp, value, datatype.

```
<reality>
  <result>OK</result>
  <node>
    <id>14</id>
    <path>/cex/40130</path>
    <quality>good</quality>
    <stamp>Thu Jun 15 12:34:03 UTC 2006</stamp>
    <value>142</value>
    <datatype>integer</datatype>
  </node>
  ...
</reality>
```

#### JSON Response

The returned hash shall contain a nodes array containing a hash for each referenced node, which includes the following keys: id, path, quality, stamp, value, datatype.

```
{
  result: "OK",
  nodes: [
    {id: 14, path: "/cex/40130", quality: 1, stamp: 1150374843.0,
```

---

```
        value: "142", datatype: 1},
    ...
  ]
}
```

## history (GET)

The history query is used to retrieve historical states for one or more nodes.

### Mandatory Parameters

URI path - the node path, or a base path (used in conjunction with the nodes parameter.)

### Optional Parameters

node/nodes/n - one or more relative or absolute node paths separated by an exclamation mark. If this parameter is included in the request, the URI path is considered as a base path for extending relative paths.

### Access Control

The client must have read permission for the specified nodes.

### XML Response (Default Flavor)

For each referenced node, a <node> tag is included in the response, containing the following tags: id, path. For each state transition, the node tag includes a <state> tag with the following sub-tags: quality, stamp, value, datatype.

```
<reality>
  <result>OK</result>
  <node>
    <id>14</id>
    <path>/cex/40130</path>
    <state>
      <quality>good</quality>
      <stamp>Thu Jun 15 12:34:03 UTC 2006</stamp>
      <value>142</value>
      <datatype>integer</datatype>
    </state>
    ...
  </node>
  ...
</reality>
```

---

## JSON Response

The returned hash shall contain a nodes array containing a hash for each referenced node, which includes the following keys: id, path. each node element also includes a states array containing state hashes with the following keys: quality, stamp, value, datatype.

```
{
  "result": "OK",
  "nodes": [
    { "path": "/cex/40130", "states": [
      { "quality": 1, "stamp": 1150374843.0,
        "value": "142", "datatype": 1},
      ...
    ]},
    ...
  ]
}
```

## set (POST)

The set query is used to update the state of a node.

### Mandatory Parameters

URI path - the node path, or a base path (used in conjunction with the nodes parameter.)

### Optional Parameters

q - the quality of the new state. This can be either a quality code or a number. The default quality is good.

s - the time stamp of the new state. If the time stamp is not specified, the server uses the current time as time stamp.

v - the new value. If the value is not specified, it is considered null. The value is ignored if the quality is unknown, bad or invalid.

dt - the datatype for the new value. This can be either a datatype code or number. If no datatype is specified, an empty datatype.

### Access Control

The client must have configure permission for the specified nodes.

### XML Response (Default Flavor)

Reality tag containing the result code.

```
<reality>
  <result>OK</result>
</reality>
```

---

## JSON Flavor

The reality hash containing the result code.

```
{"result": "OK"}
```

## write (POST)

The write query is used to write a node value. If the node is currently produced by a client, the server generates a `write_request` event for the producer. However, if the referenced node is not currently produced, the node's state is immediately updated with a forced quality.

### Mandatory Parameters

URI path - the node path, or a base path (used in conjunction with the nodes parameter.)

### Optional Parameters

`v` - the new value. If the value is not specified, it is considered null. The value is ignored if the quality is unknown, bad or invalid.

`dt` - the datatype for the new value. This can be either a datatype code or number. If no datatype is specified, an empty datatype.

`m` - the request mode. The client can set the request mode to `block`, in which the server waits until the node's value has been updated accordingly, or until a timeout period has elapsed. If the `block` mode is used, the `timeout` parameter must be specified. In the `async` mode the server does not perform any checks whether the write request has been performed by the data producer.

`t` - a timeout period in seconds for the blocking mode.

### Access Control

The client must have write permission for the specified nodes.

## XML Response (Default Flavor)

Reality tag containing the result code.

```
<reality>
  <result>OK</result>
</reality>
```

In blocking mode the server will either return a `TimeoutError` result code (which means the node's value was not updated within the timeout period), or an event containing the node's new state:

```
<reality>
  <result>OK</result>
  <event>
    <kind>state_changed</kind>
    <node_id>24</node_id>
    <node_path>/cex/40130</node_path>
```

---

```
<quality>good</quality>
<stamp>Thu Jun 15 12:34:03 UTC 2006</stamp>
<value>46</value>
<datatype>integer</datatype>
</event>
</reality>
```

### JSON Flavor

The reality hash containing the result code.

```
{"result": "OK"}
```

## open (POST)

The open query is used to open a new session. When successful, the response includes the newly created session's key.

### Mandatory Parameters

URI path - the node path (for use when mounting a producer.)

### Optional Parameters

m - whether to mount the producer at the specified node path. The value of this parameter should be yes or no.

u - the user account path.

p - the password for the user account.

### Access Control

When mounting, the client must have configure permission for the specified nodes.

### XML Response (Default Flavor)

Reality tag containing the result code and session key:

```
<reality>
  <result>OK</result>
  <skey>5a24b1823f8e74133b88f8c10e18467d</skey>
</reality>
```

### JSON Flavor

The reality hash containing the result code and session key:

```
{
  "result": "OK",
  "skey": "5a24b1823f8e74133b88f8c10e18467d"
}
```

---

## close (POST)

The close query is used to close a session.

### Mandatory Parameters

skey - the session key.

### XML Response (Default Flavor)

Reality tag containing the result code:

```
<reality>
  <result>OK</result>
</reality>
```

### JSON Flavor

The reality hash containing the result code:

```
{"result": "OK"}
```

## events (GET)

The events query is used to receive events for a session.

### Mandatory Parameters

skey - the session key.

### Optional Parameters

m - the mode to use:

normal - The server returns any pending events for the session.

block - If no events are pending, the server waits until an event is available for the session, or until the timeout period has elapsed.

stream - The server holds the connection open for the entire timeout period, adding events to the response as they occur.

t - the timeout period in seconds. The default is 60 seconds.

### XML Response (Default Flavor)

Reality tag containing the result code and session events:

```
<reality>
  <result>OK</result>
  <event>
    <kind>state_changed</kind>
    <node_id>24</node_id>
    <node_path>/cex/40130</node_path>
    <quality>good</quality>
    <stamp>Thu Jun 15 12:34:03 UTC 2006</stamp>
```



---

```
<value>46</value>
<datatype>integer</datatype>
</event>
...
</reality>
```

## JSON Flavor

The reality hash containing the result code and session key:

```
{
  "result": "OK",
  "events": [
    {"kind": 5, "path": "/cex/40130", "quality": 1,
     "stamp": 17342312.321, "value": "46", "datatype": 1},
    ...
  ]
}
```

## Node Paths in Events

Note that node paths specified in events are relative for the following events kinds: start\_track, stop\_track, update\_track, write\_request.

## touch (POST)

The touch query is used to update the timestamps of all nodes currently tracked by a producer. Each node's timestamp is updated according to the currently used sample rate for the node. Subscribing clients are notified with a state\_changed event, but the new node's state is not logged to history.

## Mandatory Parameters

skey - the session key.

## XML Response (Default Flavor)

Reality tag containing the result code:

```
<reality>
  <result>OK</result>
</reality>
```

## JSON Flavor

The reality hash containing the result code:

```
{"result": "OK"}
```

---

## signal (POST)

The `signal` query can be used to receive state events for all subscribed nodes. This query is used by consumers to make force the server to generate `state_changed` events for all subscribed nodes.

### Mandatory Parameters

`skey` - the session key.

### XML Response (Default Flavor)

Reality tag containing the result code:

```
<reality>
  <result>OK</result>
</reality>
```

### JSON Flavor

The reality hash containing the result code:

```
{reality: {result: "OK"}}
```

## sub (POST)

The `sub` query is used to subscribe to one or more nodes.

### Mandatory Parameters

URI path - the node path, or a base path (used in conjunction with the `nodes` parameter.)

`skey` - the session key.

### Optional Parameters

`node/nodes/n` - one or more relative or absolute node paths separated by an exclamation mark. If this parameter is included in the request, the URI path is considered as a base path for extending relative paths.

### Access Control

The client must have read permission for the specified nodes.

### XML Response (Default Flavor)

Reality tag containing the result code:

```
<reality>
  <result>OK</result>
</reality>
```

### JSON Flavor

The reality hash containing the result code:

```
{result: "OK"}
```

---

## unsub (POST)

The unsub query is used to cancel subscription to one or more nodes.

### Mandatory Parameters

URI path - the node path, or a base path (used in conjunction with the nodes parameter.)

skey - the session key.

### Optional Parameters

node/nodes/n - one or more relative or absolute node paths separated by an exclamation mark. If this parameter is included in the request, the URI path is considered as a base path for extending relative paths.

### Access Control

The client must have read permission for the specified nodes.

### XML Response (Default Flavor)

Reality tag containing the result code:

```
<reality>
  <result>OK</result>
</reality>
```

### JSON Flavor

The reality hash containing the result code:

```
{result: "OK"}
```

## exec (POST)

The exec query is used to issue multiple queries in a single request.

### XML Response (Default Flavor)

Reality tag containing the result code:

```
<reality>
  <result>OK</result>
</reality>
```

### JSON Flavor

The reality hash containing the result code:

```
{"result": "OK"}
```