TKN Telecommunication Networks Group

Technical University Berlin

Telecommunication Networks Group

RESTful Platform for Federating WSN Testbeds

Vlado Handziski, Claudio Donzelli, Irina Antonova

{handzisk, donzelli, antonova}@tkn.tu-berlin.de

Berlin, January, 2010

TKN Technical Report TKN-10-001

TKN Technical Reports Series

Editor: Prof. Dr.-Ing. Adam Wolisz

Note

This work has been partially supported by CONET, the Cooperating Objects Network of Excellence, funded by the European Commission under FP7 with contract number FP7-2007-2-224053. Parts of this technical report have been verbatim included in the CONET Deliverable 3.3: "Common Abstractions for Testbed Federation".

Contents

Acronyms

AAA Authentication, Authorization and Accounting

API Application Programming Interface

CO Cooperating Object

CTFS CONET Testbed Federation Server

CTF CONET Testbed Federation

CTP Collection Tree Protocol

GENI Global Environment for Network Innovations

HTTP Hypertext Transport Protocol

IC Identity Consumer

IP Identity Provider

JSON JavaScript Object Notation

MIME Multipurpose Internet Mail Extensions

MIT Massachusetts Institute of Technology

REST Representational State Transfer

RPC Remote Procedure Call

SC Service Consumer

SOA Service Oriented Architecture

SP Service Provider

SSO Single Sign-On

SUT System Under Test

TA API Testbed Adaptation API

TF API Testbed Federation API

URI Universal Resource Identifier

URN Universal Resource Name

URL Universal Resource Locator

UUID Universally Unique Identifier

XML Extensible Markup Language

Chapter 1

Introduction

The design, implementation and evaluation of Cooperating Object (CO) systems is a challenging task that is further complicated by their distributed and heterogeneous nature and the tight coupling with the environment. During the early design stages, valuable insight about the system can be gained using analytical modeling or simulation. In order to maintain tractability of the models, however, the designer is frequently forced to make simplifying assumptions about the behavior of the system components and their interaction with the environment. The associated loss of realism makes these approaches less suitable for the more advanced design stages, when the evaluation of the system performance, error resilience and other nonfunctional properties necessitate the use of real hardware, realistic environments and realistic experimental setups.

Testbeds provide convenient middle ground between simulation and full deployment on the realism axes. They allow for rigorous and controlled experimentation with the System Under Test (SUT). But similarly to full deployments, they lock the evaluation to one particular environment making it hard to differentiate between the intrinsic properties of the SUT and the effect of the specific features and external influences present at a given testbed site. One way of decoupling these influences is to cross-validate the functional and non-functional behavior of the SUT under various conditions as provided by different testbeds.

For example, Table 1.1, taken from [7], shows a summary of an experimental evaluation of the functional properties of CTP on 12 different testbeds. Experimental studies like this one are becoming indispensable part of performing credible scientific research in the area of CO. Unfortunately, their realization is currently accompanied by significant overheads in configuring the experiments and collecting the results on the individual testbeds, since easy experiment migration is hindered by a lack of common management, experiment specification and control infrastructure.

The goal of the CONET Testbed Federation (CTF) is to address some of these roadblocks by developing a software platform that will enable convenient access to the experimental resources of multiple testbeds organized in a federation of autonomous entities.

Due to the specific nature of the wireless medium, we are primarily focused on facilitating experiment migration across the federation members, and not in combining the federation resources into a single "virtual" testbed. Our federation platform is built on top of a set of common abstractions for authentication and authorization; resource discovery and reservation; and experiment specification and control. The integration APIs are light-weight, scalable

Testbed	Platform	Nodes	Physical size [m^2 or m^3]	Degree Min	Degree Max	PL	Cost	PL/Cost	Churn
Tutornet (16)	Tmote	91	$50 \times 25 \times 10$	10	60	3.12	5.91	1.9	31.37
Wymanpark	Tmote	47	80×10	4	30	3.23	4.62	1.43	8.47
Motelab	Tmote	131	$40 \times 20 \times 15$	9	63	3.05	5.53	1.81	4.24
Kanseia	TelosB	310	40×20	214	305	1.45	-	-	4.34
Mirage	Mica2dot	35	50×20	9	32	2.92	3.83	1.31	2.05
NetEye	Tmote	125	6×4	114	120	1.34	1.4	1.04	1.94
Mirage	MicaZ	86	50×20	20	65	1.7	1.85	1.09	1.92
Quanto	Epic-Quanto	49	35×30	8	47	2.93	3.35	1.14	1.11
Twist	Tmote	100	$30 \times 13 \times 17$	38	81	1.69	2.01	1.19	1.01
Twist	eyesIFXv2	102	$30 \times 13 \times 17$	22	100	2.58	2.64	1.02	0.69
Vinelab	Tmote	48	60×30	6	23	2.79	3.49	1.25	0.63
Tutornet (26)	Tmote	91	$50 \times 25 \times 10$	14	72	2.02	2.07	1.02	0.04
Blazeb	Blaze	20	30×30	9	19	1.3	-	-	-

Table 1.1: CTP evaluation on multiple testbeds published in [7]. The results illustrate how important protocol performance parameters like number of transmissions per successful delivery (Cost), average path length in hops (PL), or parent change rate (Churn) are influenced by the different testing environments.

and extensible and aim to preserve high levels of autonomy for the member testbeds in the federation. With this, we are contributing to the ongoing effort in the research community towards federated testbeds solutions in this and related areas, as exemplified by the related activities under the FIRE (EU) and the GENI (USA) initiatives.

This document presents the main features of the CTF platform. In Chapter 2, we specify the system architecture and provide short overview of the main characteristics of the REST architectural style that has been selected as guiding principle for designing the interactions between the major system components. In the next chapter, Chapter 3, we document the CTF abstractions through the identification of the main resources, their representations and the possibilities for modification of the resource state through the application of the Hypertext Transport Protocol (HTTP) method set, in accordance with the REST approach.

Chapter 2

CONET Testbed Federation Platform

The integration of CO testbeds, in themselves large heterogeneous distributed systems, is an intrinsically complex task. The CTF platform presented in this chapter should be thus considered a first step in a cyclic design refinement process. The content of the chapter is divided in two main parts. In the first, we summarize the general objectives for the CTF platform, and we provide a description of the main architectural features by focusing on the relationship between the main components and the design principles behind the core abstractions. In accordance with the selected architectural style, in the second part of the chapter, we specify the CTF platform through a detailed resource[1] model comprised of their representations and the possibilities for accessing and modifying the resource state using the HTTP method set.

2.1 Design Principles

The primary goal of the CTF is to enable convenient access to the resources of multiple CO testbeds, when organized in a loose federation of testbeds. The CTF platform provides an abstraction over the federation members and offers additional APIs that operate in the context of the federation aggregate in order to support the experimenters during the full experiment life-cycle.

In the design of the platform we have followed a set of guiding principles:

Specialization Although it shares some common characteristic with other testbed federation frameworks, the proposed CTF platform is carefully tuned to the specific needs of the CO testbeds. This specialization has enabled us to leave out some complex features that are not applicable or not important for the target domain, at the same time giving us opportunity to focus on the more important aspects like the impact of the wireless communication on the resource allocation problem, or the inclusion of tester controlled SUT mobility.

[1] Following the established practice of using the word *resource* to refer both to the real physical assets of the testbeds, as well as to the main abstract concept in REST, we apply the word in both contexts in this chapter, clarifying its meaning only when it can not be readily deduced from the surrounding text.

User-centricity In contrast to many other testbed federation frameworks that take an institution centric approach, our platform puts the individual user in the center of the design. It decouples service levels and policies from the question whether a particular user belongs to a federation member institution or not. All aspects of the service should be configurable and controllable at the granularity of a single user. This design principle, for example, has direct implications on the architecture of the AAA abstractions and has significant impact on the openness of the platform.

Scalability A typical CO testbed is a large heterogeneous distributed system which makes the task of integrating several such systems behind a common federation platform particularly challenging. Addressing these challenges requires careful engineering that leverages the best-practices learned from other successful large distributed systems, like the Web. High scalability of the solution has to be maintained by promoting stateless interactions and using caching whenever possible.

Extensibility To be successful, the CTF platform has to be kept as simple as possible, but also modular and extensible, so new features and capabilities can be organically added when they are needed. In contrast to many other testbed federation frameworks, an explicit goal of the CTF platform is the openness towards external service providers. The same APIs that are used internally to build the higher-level federation services will be also made available to external entities to promote integration with their services.

Coexistence By focusing on a set of common, testbed-independent APIs, the CTF platform will necessarily lack some specific services of the individual testbeds that are valuable to the end-users. Thus, it is crucial to enable parallel use of the native interfaces of the member testbeds and the federation APIs. The CTF platform should not limit the autonomy of the members of the federation by imposing only a single access-path to the resources of the individual testbeds.

Flexibility The usage of the CTF platform should not impose unnecessary constraints on the development process on the user side. In particular, the platform should be accessible using various programming languages and the users should have freedom in selecting the level of abstraction overhead. This should be achieved by offering a basic, language-agnostic set of API, that enables building client-side solutions for raising the level of abstraction.

In the rest of the chapter we discuss how these high level principles have been converted into concrete requirements for the overall architecture of the CTF platform and the features of the core abstractions.

2.2 System Architecture

2.2.1 Overview

To illustrate the main architectural features of the CTF platform and to establish some common vocabulary, we start by revisiting our motivating use case of performing a cross-validation experimental study on a large number of CO testbeds, introduced in Chapter 1.

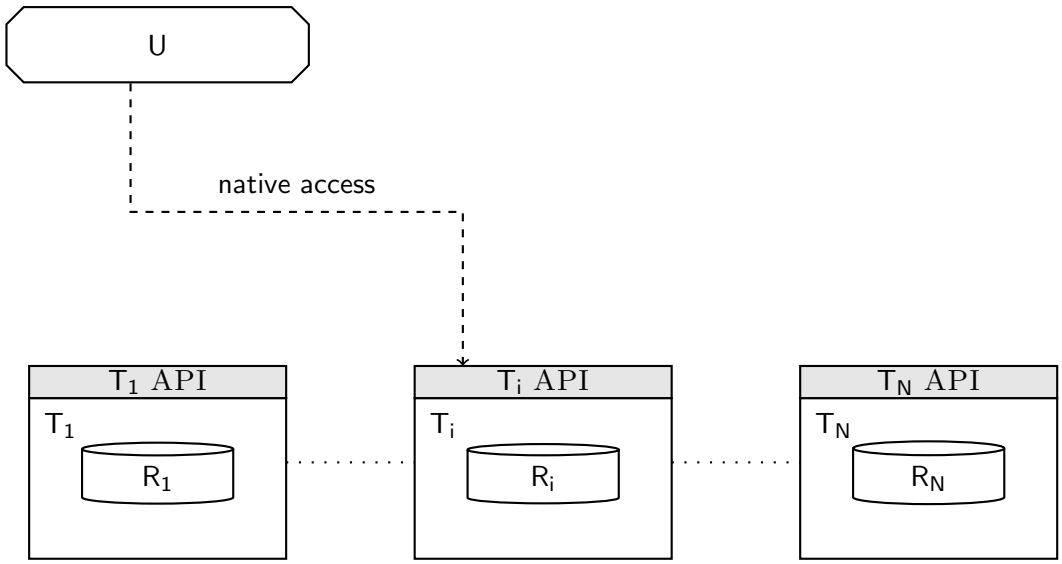

Figure 2.1: Baseline scenario S_0. The user can access the resources of the individual testbeds only over their native interfaces

Figure 2.1 depicts our baseline scenario S_0, reflecting the existing service level. The user U wants to perform a study comprised of a set of experiments $E_1, E_2, \ldots, E_N$ that need to be executed on a set of CO testbeds: $T_1, T_2, \ldots, T_N$. Each testbed provides a set of SUT resources $R_1, R_2, \ldots, R_N$ necessary for the corresponding experiment.

As it can be seen, without a federation substrate, the user can access the resources of the individual testbeds only over their native APIs: T_i API . This means that for each experiment, and after completing the native authenticating and authorization process, she needs to use a proprietary way to discover and reserve the required resources, to define and control the experiment, and finally, to collect the results.

The user needs to do this on each testbed separately. For N testbeds, she has to potentially use N different interfaces and processes. Without a federation substrate, there is no way to reuse the authentication and authorization credentials, no way to perform resource discovery across the multiple testbeds, no way to reuse the experiment specification, no way to reuse the client-side code for the on-line control of the experiment or for storing and post processing the results. In addition, there is no way to share this content with other users, so they can repeat the experiment and validate the results. There is also no common way to provide access to the results to external service providers that can provide storage or post processing (statistical analysis, plotting, etc.)

The CTF platform tries to address these limitations. Due to the large variability in the services offered by the individual testbeds this requires a two-step approach as described in the following.

As a first step, a common abstraction over the existing capabilities of the testbeds needs to be defined. This abstraction exports an interface, called *Testbed Adaptation API* (TA API), that can be used by the users to access the resources on the individual testbeds via a

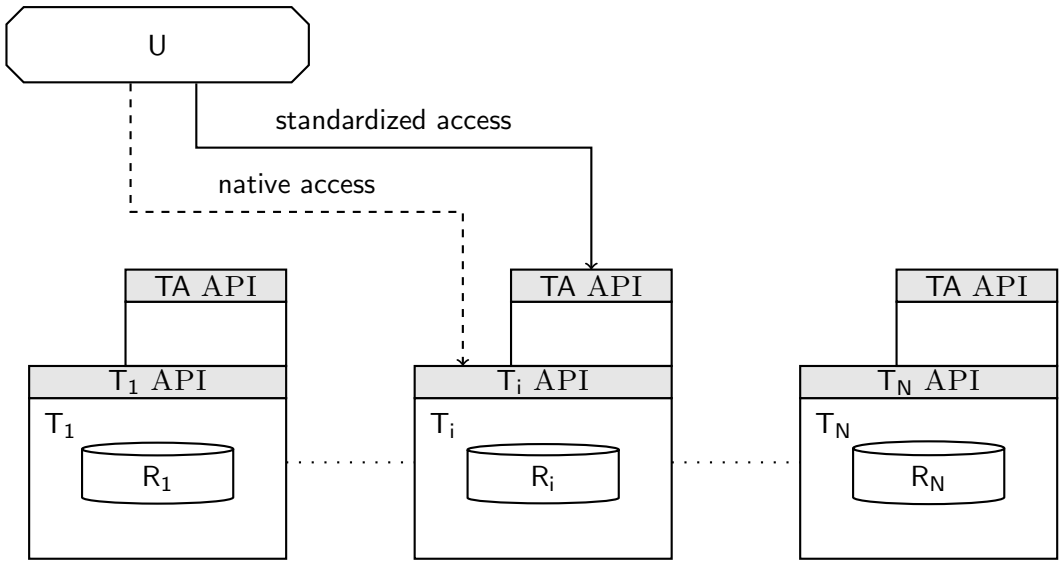

Figure 2.2: Scenario S_1. The Testbed Adaptation API enables standardized access to the resources on the individual testbeds in addition to the native interfaces.

standardized interface. In this process of interface unification, some specific features of the individual testbeds will be undoubtedly remain unrepresented. Due to this, as well as to our coexistence and autonomy principles, we expect that this common interface will be used in parallel, and not instead of the native interfaces I_{T_i}.

The TA API offers a new service level to the users, S_1, leading to the scenario is depicted on Figure 2.2. Instead of using the various native testbed interfaces I_{T_i} as in the baseline scenario, users can now access the resources over a standardized Application Programming Interface (API). When they need some testbed-specific capabilities they can always leverage the native interfaces. By incorporating adequate arbitration mechanisms, the implementation of the TA API will make sure that no conflicts in the access of the resources between the native and the federation users can occur.

Although this service level provides a significant convenience gain with respect to the S_0 scenario, by using only the Testbed Adaptation API (TA API) we are still short of providing some very useful services that abstract over the individual testbeds and operate in the context of the global federation aggregate. For example, we can provide a central place to store the definitions of the experiments, introduce a centralized discovery and reservation system, a central repository of traces and results, etc. For this we need an additional entity, a central *CONET Testbed Federation Server* (CTFS) that will offer a second interface class, the *Testbed Federation API* (TF API).

The CONET Testbed Federation Server (CTFS) leverages the services of the individual testbeds, exported through the standardized TA API, to build a higher level abstraction over the federation aggregate, thus offering new service level to the user, S_2.

Figure 2.3 illustrates the relation between the individual testbed servers exporting the native and adaptation interfaces, the federation server and the users (both native and feder-

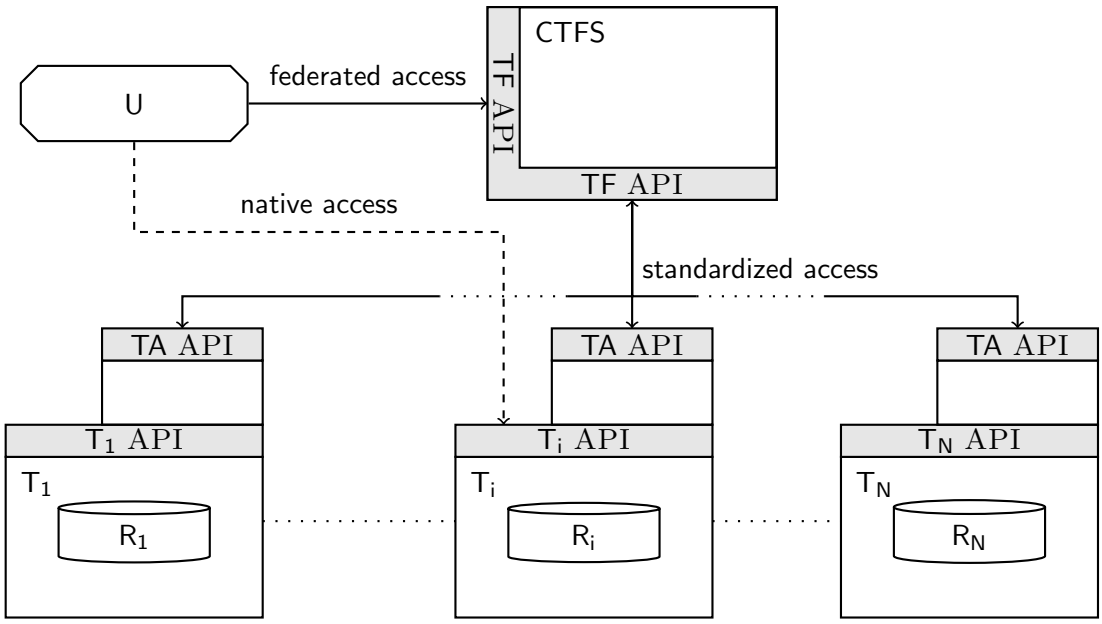

Figure 2.3: Scenario S_2. The CONET Testbed Federation Server (CTFS) leverages the services of the individual testbeds, exported through the standardized Testbed Adaptation API, to build a higher level abstraction over the federation aggregate. The new services are made available to the user over the Testbed Federation API (TF API).

ation) plus external services in the S_2 scenario. The presented architecture has the flexibility to satisfy the major requirements outlined in our motivating use-case, and represents the basis for the CTF platform solution.

2.2.2 REST Architectural Style

The overall architecture of the CTF platform can be realized using many different architectural patterns that support rich interaction between a set of distributed components. Taking into consideration the main design principles presented in Section 2.1, we have decided to base the CTF design on the *Representational State Transfer* (REST) architectural style [5].

REST is a set of design constraints for developing rich resource-oriented systems that mirror the scalability and the flexibility of the Web. In the following we provide a short overview of the core properties of systems following the REST architectural style (so called RESTful systems) and their benefits in the context of the CTF platform.

Resources

In contrast to Service Oriented Architecture (SOA) and other Remote Procedure Call (RPC)-based architectural styles where the data is kept private, encapsulated and hidden behind the processing components, in REST, the state and the nature of the data elements play a central role.

The *resource* is the main abstraction of information in a RESTful system. The *resource representation* captures the current or intended state of the resource. The components (clients, servers, etc.) act on the resources by transferring and modifying their representations. A *resource identifier* is used to uniquely identify the resource involved in the interaction.

Resource Representations

Resources are abstract entities that can only be manipulated through their representations. A resource representation is a sequence of bytes, accompanied by representation metadata that describes those bytes. The components in a RESTful system use *media types* to differentiate between the different possible representations of a resource. One scheme for specifying the media types in the system are Multipurpose Internet Mail Extensions (MIME) types [6].

The agreement on a set of common media types, together with the remaining architectural constraints, give messages in a RESTful system a "self-describing" property. Using only the media type as indication, the components in the system can safely perform many useful operations without having to look into the body of the message.

The resource model and the definition of the media type(s), used for representing the resources and driving resource and application state, forms the main cognitive effort in designing a RESTful system. Since all components in the system have to agree on the media types used for representing the resources, there is clear benefit in reusing a well defined media type from the MIME register whenever possible. Unfortunately, due to the specifics of the problem that the CTF platform is addressing, we were not able to follow this approach. Instead, we have decided to use a custom media types encoded with in a standard data interchange format.

In contrast to the existing testbed federation frameworks that use documents based on Extensible Markup Language (XML) (WISEBED uses WiseML, ProtoGENI uses Rspec, etc.), we have opted for JavaScript Object Notation (JSON) [2] as the serialization method for our resources.

JSON is more light-weight and readable then the equivalent XML serialization and is especially suitable for exchanging general data structures. Code for parsing JSON-encoded data exists for large number of programming languages. The use of JSON is also very convenient when developing rich in-browser client side applications for interacting with the CTF platform. Because all JSON text is legal JavaScript code, it is very easy for a JavaScript program to convert the serialized data into an active object. Instead of using a heavy-weight parser, like in the case of XML, one can just use the built-in `eval()` function, passing the JSON encoded representation (after a security check) as parameter.

The default media type for JSON-encoded representations is `application/json`. Following the approach used in the *Sun Cloud API* [10], for our custom JSON-based representations we use the media type designation `application/ctf.{resource}+json`, where `{resource}` stands for the name of the particular resource that is being represented.

Resource Identifiers

This ability to uniquely identify any resource in the system is a crucial characteristic of RESTful systems and represents the basis for their openness and composability. Although

REST does not impose a specific naming scheme for the resource identifiers, in practice they typically take the form of Universal Resource Identifiers (URIs), as defined in the RFC 3986 [1].

There are two major subtypes of URIs: those that also include the "location" of the resource, so that the URI can be immediately dereferenced to get the representation of the resource (i.e. Universal Resource Locators (URLs)) and those that only provide a unique name (Universal Resource Names (URNs) or other Universally Unique Identifier (UUID)-based solutions) without the location part.

The URN identifiers are more stable (for example, a URL can be rendered invalid if the domain name of the server exporting the resources is changed). Many existing testbed federation frameworks like ProtoGENI [14] and WISEBED [16], use URNs as their default identification mechanism. The URNs stability, however, comes at a cost of increased administration overheads and loss of flexibility. The URN namespaces typically need to be registered with an external register [3] like IANA [9] and one loses the standardized dereferencing capability of URL.

We believe that the benefits that URLs carry as common addressing and naming scheme outweigh their shortcomings. The CTF platform uses URLs as resource identifiers exclusively. Any problems associated with non-persistent URLs on the individual testbeds will be handled using simple URL rewriting and redirecting rules in the federation components.

Uniform interface

REST mandates a *uniform interface* between the components in the system. This is the most fundamental differentiation from the other networking styles. In contrast to the rich "verb" space in SOA and other RPC architectural styles, where each object can export different set of methods that operate on its state, in RESTful systems all resources are manipulated with exactly the same method set. In a similar way like the use of standard media types, the use of a generic interface constraints the design freedom but brings significant benefits in return.

Since all resources in the system can be manipulated with the same method set, the components don't have to implement specialized code for accessing different resources in the system. Also, the semantics of each operation is well defined and uniform across all resources and components. This leads to an interface that is easy to understand and simplifies the interoperability between large number of uncoordinated actors. The uniform method set also opens the possibility of using a standardized set of return values to inform the caller about the success of the method invocation.

In RESTful systems implemented using Web technologies, the standard HTTP method set serves the role of a uniform interface. The HTTP 1.1 standard [4] defines eight methods: OPTIONS, GET, HEAD, POST, PUT, DELETE, TRACE and CONNECT, out of which the subset: GET, PUT, DELETE and POST are most commonly used in practice. Correspondingly, the HTTP status codes serve the role of return codes.

Table 2.1 provides a succinct overview of the semantics of the most common HTTP methods when applied in a RESTful context and Section 3.1.3 illustrates the use of HTTP status codes as return values.

HTTP Method	Properties	Semantics
GET	Safe, Idempotent	Retrieve a representation of the resource identified by the Request-URI without client-relevant side effects
HEAD	Safe, Idempotent	Same as GET, but only retrieve the header information without the message-body
PUT	Idempotent	Update the resource identified by the Request-URI with the representation contained in the message-body
DELETE	Idempotent	Delete the resource identified by the Request-URI
POST	Not safe, Not idempotent	Accept the representation in the message-body as a new subordinate of the resource identified by the Request-URI; Create a new resource without knowing the final URI; Append to the state of the resource identified by the Request-URI;

Table 2.1: Uniform interface formed by the HTTP method set

The "Property" column in the table illustrates how having a common set of methods with fixed semantics that is applicable to all resources in the system also leads to a more scalable and robust system.

Issuing a HTTP GET fetches the representation of the resource identified by the URI without any "strings attached". Just like HEAD, this method is *safe*. It does not result in any change of client-relevant state at the server. Due to this properties, GET can support very efficient and sophisticated caching schemes. At the same time, the GET method is *idempotent*. The client is allowed to issue the same call one or ten times, if he wishes to do so, and the effects to the state of the identified resource at the server is guaranteed to be the same as if he made the call only once. The same idempotent property is shared by the PUT and the DELETE methods. If the client attempts to create or delete a resource and does not receive a positive status code, she knows that she can simply reissue the same request without relevant side-effects.

The HTTP POST method is neither safe nor idempotent. Making two POST requests to a collection resource will likely result in creation of two separate subordinate resources in the collection. The same applies when POST is used in a non-RESTful manner as a way to tunnel data to an arbitrary data-handling process. Because of this, the use of the POST requests has to be limited only to those scenarios where the safe or idempotent methods are not sufficient. In many cases a POST request can be avoided by reorganization of the resource model or by introducing a new resource representing the "result" of the indented activity of the original POST request.

The CTF platform uses the reduced HTTP method set described in Table 2.1 as its uniform interface. Apart from the scalability and robustness benefits explained previously, this also allows clients to cleanly separate the code that deals in generic way with the issuing of the request and the interpretation of the status codes with the service specific aspects that are part of the handling of the resource representations.

For example, fetching information about a particular node in a testbed would be implemented by a GET request on a node resource URI. The same code that handles the GET request can also be used for fetching information on a given platform, only the target will now be a platform resource URI. Similarly, the code that handles the PUT request for creating a new node resource at a given node URI can be shared with the code for creating a new binary

image at a given image resource URI.

This should be compared with classical SOA and RPC styles where there are multiplicity of methods taking the role of the uniform method set. For example, the WISEBED API has methods like `getNodeList`, `getCapabilities`, `flashImagesToSensorNodes`, etc. Every service effectively speaks a different language and both the client and the intermediary components (caches, proxies, etc.) can not make any generic assumptions about their semantic properties.

Statelessness

All REST interactions are *stateless*. In Web-based RESTful systems this means that each HTTP request should happen in full isolation and the client has to provide all of the information necessary for the server to understand the request, independent of any requests that may have preceded it.

This constraint, effectively moves the burden for maintaining the *application state* from the server to the client. The *resource state* is maintained on the server and it is the same for every client. If one client changes a particular resource state, this change is made visible to all other clients. It is the responsibility of the client to maintain any state that is specific only to that client.

The statelessness constraint frees the server from the need to retain application state between the individual requests, improving the scalability of the system. It enables parallel processing of the requests without further coordination apart from the resource state. It also allows intermediaries to view each request in isolation. For example, a cache server can make a decision whether to cache or not a result from a request without fearing that state from some previous requests might affect its validity.

Connectedness

Resource representations in RESTful systems contain links to other related resources allowing the client to *navigate* to them instead of using out-of-band information about the right URI where a given resource representation can be accessed.

A client should be able to effectively use a RESTful APIs without any prior knowledge beyond the initial URI and understanding of the standard media types appropriate for the specific application domain. From the initial URI, all application state transitions must be initiated by the client by selecting among a set of valid next states which are provided by the server as part of the representations of the manipulated resources.

Following this REST constraint, all CTF resources are interlinked and the servers guide the clients through the application state. The testbed servers remain in full control of their URI namespaces. This allows unconstrained evolution of the server-side services by limiting the effects that changes have on client-side code.

The connectedness principle is the main reason why the CTF platform is defined through its resource model and not through the URI space as usual in traditional non-RESTful Web services. As long as the client understands the media types used for representing the CTF resources, they can effectively access all services without prior knowledge of the URI hierarchy.

2.3 Core Abstractions

2.3.1 Authentication and Authorization

In order to securely identify the user, to grant the user access to all testbeds parts of the federation and to manage levels of access users should have to secured resources, the Federation of CO testbeds requires authentication and authorization facilities.

Single Sign-On (SSO) allows the user to authenticate on several servers by means of a single credential, so that the user doesn't need to use different passwords, often difficult on a large and complex network. On the other hand, we need a support for granting access to private resources according to customized policies and without exposing those resources to vulnerability.

Several protocols address the problem of authentication and authorization in computer networks. The Kerberos [11] protocol published by Massachusetts Institute of Technology (MIT), offers a solid and reliable support to authentication and authorization. Kerberos was released under open-source license and soon become a de facto standard and was adopted as a security protocol by the main operating systems.

At the time of writing, most of the applications run in the Internet and are based on the web which proved to be the answer to the increasing need of scalability. Currently, a large and growing number of web-oriented protocols address user identity trust and authentication, while much of the academic community seemed to have chosen Shibboleth [15].

Shibboleth is an institution-centric approach to identity management in which an organization is elected as a trusted verifier of the authentication data. When a user attempts to authenticate against a server supporting Shibboleth, he basically asks his organization to certify his identity. This adheres to the paradigm of delegated authentication in which both the user and the web service tightly bind to the existence of a particular institution.

This is not acceptable in our case: users should be able to control their identity over the Internet and every testbed should be given autonomy and freedom to decide how to administer access policies. For this reason we need a user-centric identity solution.

OpenID [13] is arguably the leading protocol supporting a user-centric approach and a life-long and sustainable solution to identity management. There are currently over 120 million OpenID accounts since OpenID is gaining adoption by large organizations like Google, Yahoo!, Facebook, AOL, Symantec, VeriSign, Mozilla, and Novell.

The OpenID authentication process involves three different entities: the user, the Identity Provider (IP) and the Identity Consumer (IC). When a user attempts to gain access to an IC, he is redirected to his IP. OpenID is a URL, which means that the IC can easily use it to determine the location of the IP without recourse to some kind of directory service. The IP must authenticate the user credentials and redirects the user back to the IC to allow the user access.

There are several benefits deriving from adopting OpenID. From the user's perspective, OpenID is a portable identity all over the federation and there is no need to open a new account on each of the testbeds of the federation. This also relieves each single testbed from the responsibility of storing and managing user identities while accelerating and simplifying the registration process.

While OpenID offers a light-weight HTTP-based solution to federated authentication, de-

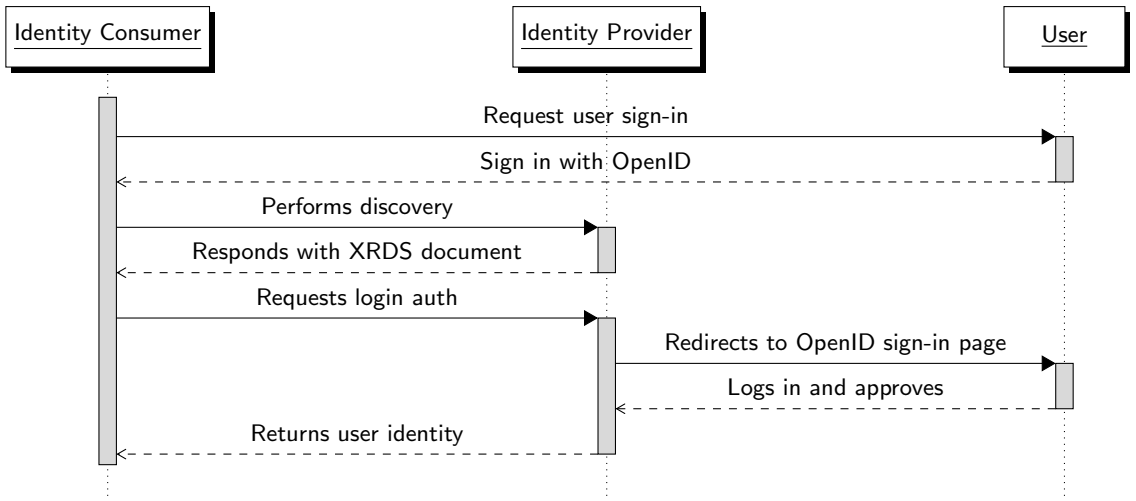

Figure 2.4: OpenID sequence diagram

signed from the same creators of OpenID, OAuth [12] addresses the problem of allowing a user to grant a first server called Service Consumer (SC) access to a specific resource stored on a second server called Service Provider (SP), without sharing any credentials with the SC. To achieve this, OAuth introduces tokens in place of user credentials. While user credentials grant unlimited access to user's private resources, tokens can restrict access to a limited set of resources or time.

The SC contacts the SP, asking for a request token. The SP verifies that the SC is registered and responds with an unauthorized request token. The SC directs the end user to an authorization page located on the SP, referencing the request token. On the authorization page, the user is prompted to log into their account and then either grant or deny limited access to their data by the SC. If the user grants access, the SC sends a request to the SP to exchange the authorized request token for an access token. The SP verifies the request and returns a valid access token. The SC sends a request to the SP. The request is signed and includes the access token. If the SP recognizes the token, it supplies the requested data.

To better understand how OpenID and OAuth meet our needs enabling authentication and authorization in the CTF platform, we describe here two use cases.

In the first use case, involving OpenID, we can imagine to provide the Federation with an OpenID server acting as an IP, implementing the OpenID protocol for third-party service authentication. The Identity Provider may be local to the Federation Server or the Federation Server itself may delegate authentication to an external existing IP, like Google [8] or AOL for example, by means of the OpenID discovery capabilities. When the user needs to access one the Testbed Servers, he needs to authenticate against them, so that they can keep a local representation of every user of the federation. To obtain this, instead of forcing the user to open a new account on every Testbed Server, we enable them to be OpenID IC.

In the second case we can imagine a user launching a job involving several resources on a specific testbed. The job physically runs on the Testbed Server but the information about the experiment specification and the document containing the experiment results should be stored in the Federation Server or in any other external storage service. To achieve this, the

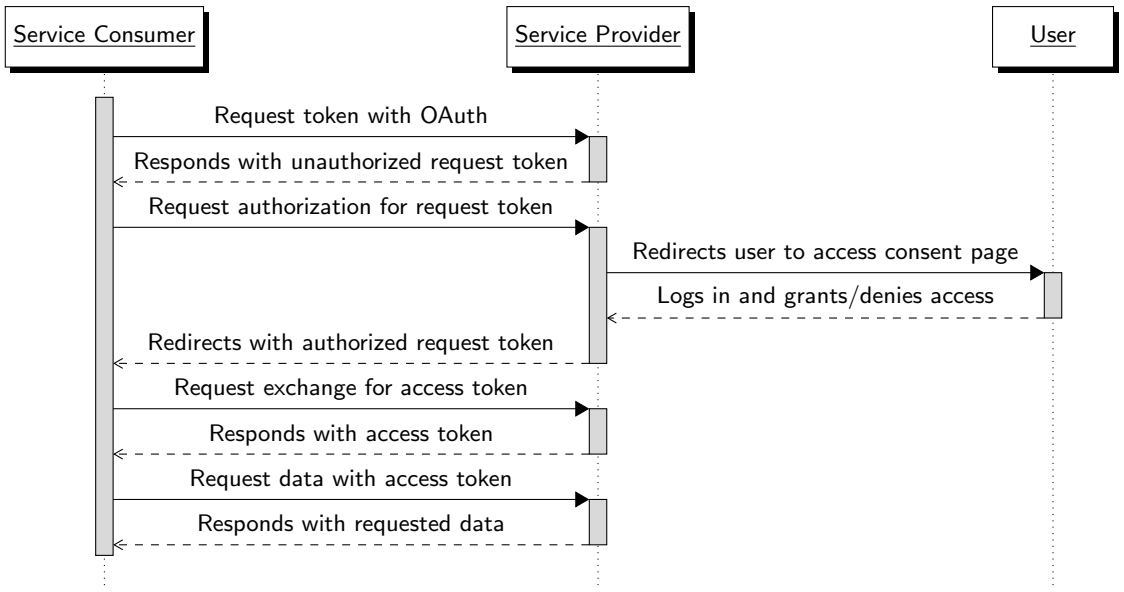

Figure 2.5: OAuth sequence diagram

user must be enabled to allow the Testbed Server to read and write his private information on his behalf. This could be obtained with an implementation of OAuth, i.e. enabling the Testbed Server to be a SC and the Federation Server to be a SP.

2.3.2 Resource Discovery, Reservation and Scheduling

As every system allowing authenticated users access to limited and distributed resources, the CTF requires discovery, reservation and scheduling capabilities, to provide the users with a view of the federation, to avoid conflicts, optimize usage and improve the user's experience.

Discovery, reservation and scheduling must be *highly available* as if they becomes unavailable the entire access to resources is compromised.

Scalability is also a strong requirement: the federation will include several testbeds, the number of users and nodes may increase considerably to potentially hundreds users and thousands of nodes and a continuous monitoring of the target infrastructure may lead to a significant load.

As we show in the next sections, Resource Discovery, Reservation and Scheduling take and inherit full advantage of the benefits of a HTTP RESTful architecture, such as identifying resources with URIs and caching. In the last section we present two notable cases of related work, emphasizing the main differences with our approach.

Resource Discovery

In a federation of testbeds, where both physical and virtual resources are distributed across a worldwide heterogeneous network, an integrated mechanism for resource discovery[2] is nec-

[2]With the term *discovery* in broad-sense we indicate here both *resource discovery* (the process of exploring/browsing for testbed resources) and the *availability discovery* (the process of retrieving information about

essary to provide users with an overview of the state of the underlying resources.

With the general term of *resources* we here indicate every single entity in the federation that can be addressed and accessed by applications. This does not include only nodes, which are the final objective of the whole CTF, but also any other entity that may take part to the experiment for any purpose, such as measuring, monitoring or simply forwarding information (i.e. channel scanners, spectrum analyzers).

While every experiment requires a specific subset of resources, we don't want to force the user to browse for them on every testbed. On the contrary we need to provide the user with an abstract aggregated view of the current state of the federation through a minimum number of effort and a priori assumptions.

Moreover, resource discovery must allow tracking of both relatively static and frequently changing resource characteristics and loosening and tightening of the granularity of the searching filter, according to the user's needs, by means of an expressive and adaptive query language.

Discovery happens at two different levels: *federation level* and *testbed level*. At *federation level* discovery is in charge to gather information coming from the different testbeds, while at *testbed level* the information is retrieved from the testbed resources. Such a hierarchical structure allows the discovery query processing to be distributed among all testbeds avoiding significant load at the federation level.

A discovery query issued to the CTFS may be split in *resource discovery* and *availability discovery*. The *resource discovery* answers to the question *What is present?* while *availability discovery* answers to *What is available?*. Since the result of *resource discovery* is likely slowly changing over time, it can be cached in the CTFS, behaving here like a shared cache (proxy). During *availability discovery* the system retrieves a collection of which resources are currently available in a specified time slot for a specified user. No detail about resources is returned in this phase, but only whether each resource is available for use or not. This is context-sensitive as it might be subject to access and priority policies. Decoupling information about resources and their availability allows discovery to inherit full advantage of REST, allowing optimization of caching.

This discovery request contains information about the searching parameters that could appear as HTTP GET query parameters or be uploaded (HTTP POST) as a document describing the experiment to be performed including the needed resources.

An example list of parameters follows here:

- *platform type*
- *radio technology*
- *amount of nodes per node type*
- *duration of experiment*
- *mobile node required*
- *sensors*

resources *free* from other users reservations). Later in this section we will show how this decoupling can improve scalability.

- *actuators*

Resource Reservation

In order to avoid collision and to allow concurrent access at same time, the CTF includes a global resource reservation system.

With collision, we mean the conflict generated by concurrent access to the a shared portion of space, frequency and time. In our case, if we assume that the different testbeds are far apart enough to grant no electromagnetic interference during concurrent experiments, we can conclude that collision is avoided as long as experiments run on different time slots and on different portion of the spectrum. A first simple approach would consist in reserving an entire testbed for a given slot of time. This would be enough to avoid any kind of conflicts during concurrent access to the federation by different users, but will not contribute to optimal usage of the resources.

In line with the federation philosophy, none of the resource reservation features should impose significant limits in autonomy and functionality of the individual testbeds. We assume that every testbed in the federation is already capable of managing reservations through the TA API. Where the native API does not allow concurrent access, we introduce a locking system at the *federation level* which temporarily isolates the testbed from the federation during access by native users.

Resource Scheduling

The CTFS introduces another layer of optimization which strongly impacts the user experience. A complete knowledge of the status of the underlying resources allows the CTF to support additional functions like solving dependent reservation requests featuring inter-testbed parameters and optimizing the usage of resources by providing the user with a series of alternatives (both in resources and in time) whenever the request for reservation could not succeed.

In the ProtoGENI implementation and deployment of Global Environment for Network Innovations (GENI), a virtual laboratory for at-scale networking experimentation, RSpec defines concepts like *advertisements*, *requests* and *tickets* which constitute an ad-hoc solution to the problem of discovery and reservation. While *Advertisements* describe resources provided by component managers, *requests* describe the resources that a client wants and *tickets* are resource reservations which are promised by a component manager to a client or broker. In our case *advertisements* are simply responses to aggregated requests about the state of the federation (mostly related with slowly varying information), *requests* are simply discovery queries and *tickets* are simply URLs pointing at the resource created out of a successful request for a new reservation.

2.3.3 Experiment Specification and Control

The main goals of the CTF is to facilitate running a given experiment on different testbeds. To do this conveniently the CTF must provide facilities for specifying the experiment setup in a testbed independent manner, as well as a facility for issuing the control commands necessary for preparing the experiments environment, executing the experiment and collecting results.

Experiment Specification

An experiment that is to be run on CTF is defined in a generic way so that it can be reused on all testbeds participating in the federation. It must define the types of hardware and software resources, as well as other properties of the infrastructure of the testbed in which the experiment should be run. The static types of hardware and software resources are described independently from the participating testbed resources in form of HTTP REST resources. Thus new types of device resources can easily be added to the federation by creating new resources without interfering with resources already in use. Each experiment contains one or more *property sets* that define what kind of devices are needed and how many of them. Each property set defines a unique combination of properties of a node type needed in the experiment. Nodes must have at least these properties but may offer other properties that are not defined in the property set. The node count in each property set defines how many of those nodes are needed, leading to disjoint property sets in the sense that when creating a property set with node count n, it is imperative to create exactly n virtual nodes that reference this property set. This part of the experiment specification is matched with the available resources from resource discovery.

The second part that needs to be specified are the tasks of the experiment, their order of execution and on which nodes they have to be executed. When executing an experiment there are two options:

In *batch mode* the user specifies all necessary tasks and setup for experiment execution in the experiment definition. In this case no further user interaction is necessary. The experiment is executed and the results are stored.

In *interactive mode* a subset of the experiment tasks are specified in the experiment definition, but the user may interact with experiment execution during runtime by adding further tasks on the fly. This allows the user to dynamically modify the course of the experiment.

This approach allows reusing experiments as templates. Thus by modifying part of an existing experiment(adding, removing, modifying or rescheduling tasks or virtual node groups) the user can easily create new experiments.

There are 2 kinds of task and node resources: Virtual tasks, virtual nodes(forming virtual node groups) and tasks and nodes (forming node groups). The former are needed to define the experiment independently of the executing testbed.

Virtual nodes are needed for naming each node participating in the experiment. Each virtual node must corresponds to exactly one real node found by resource discovery. So when a task is defined a reference on which nodes it needs to be done can be added. Some tasks need to be done at the same time on several different virtual nodes, so these virtual nodes are grouped forming *virtual node groups*. A virtual node holds a name and is part of a property set. A virtual node group is formed for grouping all nodes that need to be addressed for executing one virtual task at a certain point in time. That is why a virtual node group may contain virtual nodes of different property sets. The characteristic grouping the nodes is the task and not the properties of the nodes. A consequence of this is that virtual nodes in a virtual node group are neither homogeneous nor exclusively found in one group. Consequently virtual node groups are not necessarily disjoint.

Virtual tasks describe which steps exactly have to be executed for the experiment. Each experiment step must define time of execution(relative to start of the experiment), which

operation has to be executed, which nodes are involved and which input is needed and where it is located. As we use the REST architectural style, all information needed is stored within the federation in form of referenced resources or directly within the task resource. Besides some administration fields the virtual tasks resource contains the following fields:

- *eta - time of execution of task, relative to start time*
- *method - Verb defining type of HTTP request*
- *headers - Headers defining media type/representation of reply and request, cache options, authorization options, definition of host*
- *payload - JSON representation of the needed input information for this operation, may contain references to other resources of fixed name-value pairs involved in the operation*
- *target - The reference to the virtual node group resource that references all virtual nodes that participate in this task.*

The actual task is then constructed by assembling those fields into an HTTP request that is issued to the real node group. As the virtual task is constructed for the purpose of defining an experiment step independent from any testbed, the virtual task will not be translated into an HTTP request but is used as a generic variant describing what operation needs to be executed on what kind of nodes. This approach allows us to be independent of a fixed set of operations/experiment tasks or node types. Thus the experiment specification can be extended at any point without having to change existing federation resource types and can be used to create a testbed dependent experiment description by mapping the virtual node groups to testbed dependent node groups.

An example set of experiment tasks:

- *load image on node*
- *load image on node_group*
- *reboot node*
- *reboot node_group*
- *delete image from node*
- *delete image from node_group*
- *move a node*
- *move a node_group*
- *inject message on node*
- *inject message on node_group*
- *extract message on node*

- *extract message on node_group*
- *start battery-power node*
- *stop battery-power node*
- *start usb-power node*
- *stop usb-power node*
- *start monitoring node*
- *stop monitoring node*
- *start tracing*
- *stop tracing*
- *pick trace*

Experiment Control

After having specified the experiment setup, we need to define how the experiment specification can be translated into a testbed dependent experiment version. And what is needed on the testbed-side for being able to translate the experiment tasks coming from the CTF into native testbed tasks that are executed at the time defined in the experiment.

Virtual tasks cannot be executed. When a matching testbed is found and a job has been scheduled the user is sent references(URI) to the reserved nodes. Those nodes must have a one to one mapping to the virtual nodes. Node groups corresponding to the virtual node groups must be created as well. The testbed-dependent task is then created by getting the virtual task representation and replacing the virtual node group references with the real node groups references. Those testbed dependent tasks are then processed by the CTFS. The CTFS translates each of the tasks into HTTP requests that are sent to the testbed server. Each testbed server must implement the TA API. Thus the tasks coming from the CTFS are understood by the testbed and queued in a tasks execution engine. According to the scheduling of the tasks received by the CTFS will the task execution engine trigger executing the testbed native tasks. The initial request of the CTFS receives a response with HTTP status code 202(see Section 3.1.3) and a reference to a status resource. Completion of the request is then announced to the status resource that can be polled for further information.

2.3.4 Logging and Tracing

The CTF must provide a facility for logging and storing information about events that occur while running an experiment. There are two kinds of events. Events that reflect changes in the testbed setup during an experiment and events that are directly triggered by the SUT. The former are referred to as logs, the latter as traces.

Experiment control on the CTFS is done by issuing HTTP requests to dedicated target resources, e.g. node groups. The change in the state of a resource of the CTF reflects changes in the testbed setup. Those changes need not only be reported to the tasks resource that

caused them, but are also logged by creating a representation of a log resource. Similar to the concept of the OASIS Base event, where a predefined state change of a source component is reported by another component the CTF introduces the Log resource. The Log resource binds the finishing of a task on a dedicated resource to its status resource and adds a timestamp.

The Trace resource is triggered by the SUT, by the image that is burned on the nodes under test. As the CTF federates testbeds of different infrastructure it must be able to deal with nodes providing their traces via heterogeneous conduits and protocols. In order to be able to create a homogeneous trace files the nodes of each testbed provide information via what type of interface the output is provided. Besides the trace content the trace resource contains reference to the job, testbed, project and user.

Chapter 3

CONET Testbed Federation Platform APIs

This section specifies the RESTful API of the CTF platform. The structure of the specification is inspired by the *Sun Cloud API* [10], a public Application Programming Interface (API) for Cloud services that most closely follows the spirit of the REST architectural style.

3.1 Common Features

This section specifies constraints that apply to all requests and responses in the CTF API. It includes the common headers, with special focus on the fields enabling efficient caching, and the shared status codes according to the HTTP /1.1 specification (RFC 2616) [4].

3.1.1 HTTP Request Headers

Requests issued to the CTF APIs should include the following HTTP headers.

Header	Description
Accept	Specifies which media type(s) are acceptable for the response.
Authorization	HTTP Basic Authentication credentials, in case of access to protected resources by the user or by any other authorized entity.
Cache-Control	Specifies the caching directive(s) that must be followed in the request-response chain.
Content-Length	Indicates the size of the request body.
Content-Type	Indicated the media type of the request body.
Host	Specifies the Internet host and port number of the requested resource.
If-None-Match	Specifies a condition on the Etag for the request to be processed: the request should be forwarded to the server only if the cached response has been different ETag (GET, HEAD).
If-Match	Specifies a condition on the Etag for the request to be processed: the request should be forwarded to the server only if the cached response has not been modified (PUT, DELETE).
If-Modified-Since	Specifies a condition on the time for caching: the request should be forwarded to the server only if the cached response has been modified (GET, HEAD).
If-Unmodified-Since	Specifies a condition on the time for caching: the request should be forwarded to the server only if the cached response has not been modified (PUT, DELETE).

3.1.2 HTTP Response Headers

Responses returned by the CTF APIs should include the following HTTP headers.

Header	Description
Allow	Lists the set of methods supported by the resource identified by the Request-URI.
Cache-Control	Specifies the caching directive(s) that must be followed in the request-response chain.
Content-Length	Indicates the size of the response body.
Content-Type	Indicated the media type of the response body.
ETag	Provides the current value of the entity tag for the requested variant.
Expires	Gives the date/time after which the response is considered stale unless it is first validated with the origin server.
Last-Modified	Indicates the date and time at which the origin server believes the variant was last modified.
Location	This field is used to redirect the recipient to a location other than the Request-URI for completion of the request or identification of a new resource.
WWW-Authenticate	This field must be included in 401 (Unauthorized) response messages to invite the client to submit credentials for accessing a protected resource.

3.1.3 HTTP Status Codes

The CTF APIs will return HTTP status codes as described in the following table.

HTTP Status	Description
200 OK	The request was successfully completed.
201 Created	A request that created a new resource was completed, and no response body containing a representation of the new resource is being returned. A Location header containing the canonical URI for the newly created resource should also be returned.
202 Accepted	The request has been accepted for processing, but the processing has not been completed. The response will include an indication of the request's current status, and either a pointer to a status monitor or some estimate of when the user can expect the request to be fulfilled.
204 No Content	The server fulfilled the request, but does not need to return a response message body.
304 Not Modified	If the client has performed a conditional GET request and access is allowed, but the document has not been modified, the server will respond with this status code.
400 Bad Request	The request could not be processed because it contains missing or invalid information (such as validation error on an input field, a missing required value, and so on).
401 Unauthorized	The authentication credentials included with this request are missing or invalid.
403 Forbidden	The server recognized your credentials, but you do not possess authorization to perform this request.
404 Not Found	The request specified a URI of a resource that does not exist.
405 Method Not Allowed	The HTTP verb specified in the request (DELETE, GET, HEAD, POST, PUT) is not supported for this request URI.
406 Not Acceptable	The server is refusing to service the request because the Request-URI is longer than the server is willing to interpret.
409 Conflict	A creation or update request could not be completed, because it would cause a conflict in the current state of the resources supported by the server (for example, an attempt to create a new resource with a unique identifier already assigned to some existing resource).
410 Gone	The requested resource is no longer available at the server and no forwarding address is known. This condition is expected to be considered permanent. Clients with link editing capabilities SHOULD delete references to the Request-URI after user approval.
412 Precondition Failed	The precondition given in one or more of the request-header fields evaluated to false when it was tested on the server.
415 Unsupported Media Type	The resource identified by this request is not capable of generating a representation corresponding to one of the media types in the Accept header of the request.
500 Internal Server Error	The server encountered an unexpected condition which prevented it from fulfilling the request.
501 Not Implemented	The server does not (currently) support the functionality required to fulfill the request.
503 Service Unavailable	The server is currently unable to handle the request due to temporary overloading or maintenance of the server.

3.1.4 Common Resources

Status [application/ctf.Status+json]

A Status represents a report on the progress of an asynchronous request for a change in the state of a specific resource. Such requests receive an HTTP status code of 202 Accepted, with the response body being a Status representation.

Status resource model contains the following fields:

Field Name	Type	Occurs	Description
uri	URI	1	A GET against this URI refreshes the representation of this resource.
name	String	1	A given name describing the request whose status this represents.
media_type	String	1	`application/ctf.Status+json`
num_done	Integer	1	Represents the progress towards completion of the request. A value equal to *num_tot* indicates that the request has completed.
num_tot	Integer	1	Reprents the value that *num_done* reaches when the request had completed.
target	URI	1	Represents the resource upon which the request is acting.
message	String	0..1	Brief message describing the completed operation (if successful) or an error message (if not successful).

Index [application/ctf.Index+json]

An Index represents a collection of resources. It allows browsing and creation of new resource by submitting an HTTP POST request to this resource.

The Index resource model contains the following fields:

Field Name	Type	Occurs	Description
uri	URI	1	A GET against this URI refreshes the representation of this resource.
name	String	1	A given name for this resource.
media_type	String	1	`application/ctf.Index+json`
items	Resource[]	1	An array of URIs to resources belonging to this collection.

3.2 Resource Model

According to the REST style, this section specifies the resources comprising the CTF platform. The following table provides an overview of the resources and their association with the TF API and the TA API:

Part of TF API	Part of TA API and TF API
Federation	Testbed
Project	User
Experiment	Job
PropertySet	NodeGroup
VirtualNodeGroup	Image
VirtualNode	Task
VirtualTask	Trace
	Log
	Socket
	Node
	Platform
	Interface
	Radio
	Sensor
	Actuator
	Mobility
	ImageFormat

3.2.1 Testbed Federation API

Federation [application/ctf.Federation+json]

A Federation represents a user's view of all accessible Testbeds, Users and Projects in the Federation of Testbeds. It is also the starting point for discovery of all other resources within the Testbed Federation API. Furthermore it includes common resource types(Platforms, Sensors, Actuators, Interfaces) that should be universally used when describing the properties of the testbed resources.

The Federation resource model contains the following fields:

Field Name	Type	Occurs	Description
uri	URI	1	A GET against this URI refreshes the representation of this resource.
name	String	1	A given name for this resource.
media_type	String	1	`application/ctf.Federation+json`
users	User[]	0..1	The list of Users currently registered to the CONET Testbed Federation.
projects	Project[]	0..1	The list of Projects currently stored in the CONET Testbed Federation.
testbeds	Testbed[]	0..1	The list of Testbeds currently registered to the CONET Testbed Federation.
platforms	Platform[]	0..1	The list of platforms adopted for nodes within the CONET Testbed Federation (testbeds supporting platforms must refer to this resource).
radio_technologies	RadioTechnology[]	0..1	The list of radio technologies adopted for nodes within the CONET Testbed Federation (nodes with radio devices must refer to this resource).
sensors	Sensor[]	0..1	The list of sensors adopted for nodes within the CONET Testbed Federation (nodes with sensors must refer to this resource).
actuators	Actuator[]	0..1	The list of actuators adopted for nodes within the CONET Testbed Federation (nodes with actuators must refer to this resource).
interfaces	Interfaces[]	0..1	The list of interfaces defined for communication with nodes within the CONET Testbed Federation (nodes implementing interfaces must refer to this resource).
image_formats	ImageFormat[]	0..1	The list of software image file formats within the CONET Testbed Federation (nodes allowing burning image files must refer to this resource).

Project [application/ctf.Project+json]

A Project represents a grouping of Experiments used in its context. It also holds references to Users participating in the project by running, designing and evaluating experiments.

Project resource model contains the following fields:

Field Name	Type	Occurs	Description
uri	URI	1	A GET against this URI refreshes the representation of this resource.
name	String	1	A given name for this resource.
media_type	String	1	`application/ctf.Project+json`
description	String	1	A short description of the Project.
users	Users[]	1	A list of Users registered to this Project.
testbeds	Testbed[]	0..1	A list of Testbeds the Project acts on.
experiments	Experiment[]	0..1	A list of Experiments used.
jobs	Job[]	0..1	A list of Jobs that implement project-related experiments on the federated testbeds.

Experiment [application/ctf.Experiment+json]

The Experiment resource represents an experiment in terms of owner (User), documentation (Title, Description), the list of Jobs which implement the Experiment on different Testbeds, and finally the output data collected from all related Jobs. Experiments have three different sharing modes: *public*, *protected* and *private*. *Public* experiments are accessible by all users in the Federation, *protected* experiments are accessible by all users within the same Project and *private* experiments are only accessible to the Experiment's owner (the User). All experiment resources contain a reference to the project they are allocated to.

Field Name	Type	Occurs	Description
uri	URI	1	A GET against this URI refreshes the representation of this resource.
name	String	1	A given name for this resource.
media_type	String	1	`application/ctf.Experiment+json`
description	String	1	A short description of the Experiment.
owner	User	1	User owning the experiment.
project	Project	1	Project the experiment is allocated to.
testbeds	Testbed[]	0..1	A list of testbeds the experiment has been implemented for.
image_files	Image[]	0..1	A list of image files needed for running the experiment.
property_sets	PropertySet[]	0..1	Array of all property sets that define the kind of devices needed for experiment execution.
virtual_tasks	VirtualTasks[]	0..1	Array of all virtual tasks, that are needed for experiment execution.
virtual_node_groups	VirtualNodeGroup[]	0..1	Array of all virtual node groups needed for executing virtual tasks.
virtual_nodes	VirtualNode[]	0..1	Array of all virtual nodes needed for executing virtual tasks.
jobs	Job[]	0..1	A list of job that have been scheduled for the experiment.
traces	Trace[]	0..1	A list of Traces created when running the experiment-related jobs.
sharing	String	1	String indicating the sharing mode of the experiment. Either one of: *public*, *protected* and *private*.

PropertySet [application/ctf.PropertySet+json]

The PropertySet resource represents a grouping of all properties required by a node for an experiment. It contains information about the needed platform type, interface types, sensor

and actuator types, radio technology and the number of nodes needed with these properties. This information is used during the resource discovery process to locate testbeds capable of executing the experiment.

The PropertySet resource model contains the following fields:

Field Name	Type	Occurs	Description
uri	URI	1	A GET against this URI refreshes the representation of this resource.
name	String	1	A given name for this resource.
media_type	String	1	`ctf.PropertySet+json`
description	String	1	A short description for this resource.
owner	User	1	The user owning this property set.
experiment	Experiment	1	Experiment that uses this property set.
platform	Platform	0..1	The platform type needed.
radios	Radio[]	0..1	The type of radio technology required by the experiment.
interfaces	Interface[]	0..1	The Interface types that are required by the experiment.
sensors	Sensor[]	0..1	Sensors that are required by the experiment.
actuators	Actuator[]	0..1	Actuators that are required by the experiment.
mobility	Mobility	0..1	The supported type of mobility
node_count	Integer	1	The number of nodes needed of this type.

VirtualNodeGroup [application/ctf.VirtualNodeGroup+json]

The VirtualNodeSet resource represents a frozen set of nodes that can be created for different purposes. It is a subset of the nodes participating in an experiment. A virtual node set is composed of all virtual nodes that take part in one task, e.g. switch off power supply to all nodes in this virtual node set. Virtual node groups are not necessarily disjoint.

The VirtualNodeSet resource model contains the following fields:

Field Name	Type	Occurs	Description
uri	URI	1	A GET against this URI refreshes the representation of this resource.
name	String	1	A given name for this resource.
media_type	String	1	`application/ctf.VirtualNodeGroup+json`
virtual_nodes	VirtualNode[]	1	A list of virtual nodes.

VirtualNode [application/ctf.VirtualNode+json]

The VirtualNode resource allows naming of each node participating in the experiment. Naming is necessary for defining which virtual tasks have to act on which dedicated virtual nodes. Furthermore the virtual node references the property set that defines its features.

The VirtualNode resource model contains the following fields:

Field Name	Type	Occurs	Description
uri	URI	1	A GET against this URI refreshes the representation of this resource.
name	String	1	A given name for this resource.
media_type	String	1	`application/ctf.VirtualNode+json`
property_set	PropertySet	1	The property set defining the properties of this virtual node.

VirtualTask [application/ctf.VirtualTask+json]

A virtual task describes one step of the experiment execution. A Task can be loading an image to a single node or a whole node group, erasing an image from a single node or a whole node group, starting or stopping tracing, powering a node or node group on or off. Information about which of the named generic tasks should be executed is defined by the triple method, target_uri and payload. The virtual task differs from the task, as it only operates on virtual node sets.

Virtual Task resource model contains the following fields:

Field Name	Type	Occurs	Description
uri	URI	1	A GET against this URI refreshes the representation of this resource.
name	String	1	A given name for this resource.
media_type	String	1	`ctf.VirtualTask+json`
description	String	1	Short description of the virtual task.
eta	Integer	1	Integer value representing time of execution of task, relative to start time of the job.
method	String	1	Verb defining type of HTTP request.
headers	String	1	Headers defining media type/representation of reply and request, cache options, authorization options, definition of host.
payload	String	1	JSON representation of the needed input information for this operation, may contain references to other resources of fixed name-value pairs involved in the operation.
target	String	1	The URI of the virtual node group resource on which this request should act on.

3.2.2 TA API and TF API

Testbed [application/ctf.Testbed+json]

The Testbed resource is the representation of a real testbed member of the CONET Testbed Federation.

Testbed resource model contains the following fields:

Field Name	Type	Occurs	Description
uri	URI	1	A GET against this URI refreshes the representation of this resource.
name	String	1	A given name for this resource.
media_type	String	1	`application/ctf.Testbed+json`
organization	String	1	The name of the organization running this testbed.
sockets	Socket[]	0..1	The list of sockets that may host nodes in this testbed.
nodes	Node[]	0..1	The list of nodes connected to the testbed.
platforms	Platform[]	0..1	Platforms currently supported by this testbed.
sensors	Sensor[]	0..1	The list of Sensors supported for Nodes within the testbed.
actuators	Actuator[]	0..1	The list of Actuators supported for Nodes within the testbed.
interfaces	Interfaces[]	0..1	The list of Interfaces supported for communication with Nodes within the testbed.
jobs	Job[]	0..1	Jobs scheduled on the testbed.

User [application/ctf.User+json]

A User represents the user registered to the CONET Testbed Federation. The User resource model references all projects and experiments owned by the user. Furthermore it contains references to all experiments run by the user in form of references to jobs and traces.

User resource model contains the following fields:

Field Name	Type	Occurs	Description
uri	URI	1	A GET against this URI refreshes the representation of this resource.
name	String	1	A given name for this resource.
media_type	String	1	`application/ctf.User+json`
openid	String	1	OpenID URL belonging this User.
email	String	1	E-mail address for the User.
organization	String	1	Organization which the User is part of.
projects	Project[]	0..1	Projects the User is involved.
experiments	Experiment[]	0..1	The Experiments owned by the User.
jobs	Job[]	0..1	Jobs owned by the User.

Job [application/ctf.Job+json]

In case of *batch mode*, the user must specify every single Task to be automatically performed, including boot, configuration, control, tracing and shut-down. In case of *interactive mode*, the tasks attribute includes only the common Tasks needed for the boot and the shut-down. All other Task are left to the user.

Job resource model contains the following fields:

Field Name	Type	Occurs	Description
uri	URI	1	A GET against this URI refreshes the representation of this resource.
name	String	1	A given name for this resource.
media_type	String	1	`application/ctf.Job+json]`
description	String	1	Short description for this resource.
owner	User	1	The user running this job.
project	Project	1	The Project this job is related to.
experiment	Experiment	1	The Experiment that was implemented by this job.
testbed	Testbed	1	The testbed on which this job is to be run.
platform	Platform	1	The platform which this job will use.
datetime_from	Timestamp	1	The time when the Job starts.
datetime_to	Timestamp	1	The time when the Job shuts down.
duration	Integer	1	The duration of this job expressed in minutes.
nodes	Node[]	1	The list of nodes involved in this job.
node_groups	NodeGroup[]	0..1	Frozen sets of nodes involved in this job.
tasks	Task[]	0..1	The list of operations that has to be performed during this job.
traces	Trace[]	0..1	The list of trace files created during the execution of the job.

NodeGroup [application/ctf.NodeGroup+json]

A NodeGroup is a frozen set of Nodes that can be created for different purposes.

NodeGroup resource model contains the following fields:

Field Name	Type	Occurs	Description
uri	URI	1	A GET against this URI refreshes the representation of this resource.
name	String	1	A given name for this resource.
media_type	String	1	`application/ctf.NodeGroup+json`
nodes	Node[]	1	A list of Nodes.

Image [application/ctf.Image+json]

The Image resource provides an image ready to be burned on a node. Furthermore it defines its own format and symbols that allow generic variables in the image file to be replaced with values that are provided only at runtime.

Image resource model contains the following fields:

Field Name	Type	Occurs	Description
uri	URI	1	A GET against this URI refreshes the representation of this resource.
name	String	1	A given name for this resource.
media_type	String	1	`Image [application/ctf.Image+json]`
description	String	1	A short description for this Image.
owner	User	1	The user that owns this Image.
image_format	ImageFormat	1	Image format as defined by the ImageFormat resource.
content	Binary	1	The raw content of the Image file.
mapping_keys	String[]	0..1	A list of symbols in the Image file allowing generic variables to be assigned at runtime.
mapping_values	String[]	0..1	A list of values to assign at runtime to respective symbols.

Task [application/ctf.Task+json]

A Task describes one step of the experiment/job execution. A Task can be loading an image to a single node or a whole node group, erasing an image from a single node or a whole node group, starting or stopping tracing, powering a node or node group on or off. Information about which of the named tasks should be executed is defined by the triple method, target_uri and payload. The Field relative_time defines the time to execute the task relative to the beginning of a Experiment/Job start time.

Task resource model contains the following fields:

Field Name	Type	Occurs	Description
uri	URI	1	A GET against this URI refreshes the representation of this resource.
name	String	1	A given name for this resource.
media_type	String	1	`application/ctf.Task+json`
description	String	1	Short description of the task.
eta	Integer	1	Integer value representing time of execution of task, relative to start time of the job.
method	String	1	Verb defining type of HTTP request.
headers	String	1	Headers defining media type/representation of reply and request, cache options, authorization options, definition of host.
payload	String	1	JSON representation of the needed input information for this operation, may contain references to other resources of fixed name-value pairs involved in the operation.
target	String	1	The URI of the node group resource on which this request should act.
status	Status	0..1	The URI of the status resource containing information about the execution state of the task request.

Trace [application/ctf.Trace+json]

The Trace resource is a resource that contains the results of running an experiment on a dedicated testbed. The Trace resource holds references to the job that produced the result, the testbed where it was run as well as the experiment that was implemented and the project it is related to. The Content field contains the actual output that was collected when the Job was run.

Trace resource model contains the following fields:

Field Name	Type	Occurs	Description
uri	URI	1	A GET against this URI refreshes the representation of this resource.
name	String	1	A given name for this resource.
media_type	String	1	`application/ctf.Trace+json`
description	String	1	Short description of the trace.
user	User	1	User owning the trace.
project	Project	1	Project in which context the trace was created.
experiment	Experiment	1	The Experiment that was implemented, run an that produced the trace.
testbed	Testbed	1	Testbed on which the trace was generated.
job	Job	1	Job that created the trace file.
content	String[]	0..1	Each String is composed of the timestamp when the message was issued, the node it was issued by, the actual message payload and the number of bytes of the message payload.

Log [application/ctf.Log+json]

The Log resource links information about a change of the state of a component of the testbed, by referencing a task, its target and the status and the time of finished execution.

Log resource model contains the following fields:

Field Name	Type	Occurs	Description
uri	URI	1	A GET against this URI refreshes the representation of this resource.
name	String	1	A given name for this resource.
media_type	String	1	`application/ctf.Log+json]`
job	Job	1	The job that caused this Log.
task	Task	1	Task that was logged.
status	Status	1	Status resource that reported the finishing of the task.
target	URI	1	URI of the resource changed.
timestamp	Integer	1	Time when the event logged occurred.

Socket [application/ctf.Socket+json]

The Socket represents a testbed socket which serves as a container for information about a testbed receptacle that is independent from the node.

Socket resource model contains the following fields:

Field Name	Type	Occurs	Description
uri	URI	1	A GET against this URI refreshes the representation of this resource.
name	String	1	A given name for this resource.
media_type	String	1	`application/ctf.Socket+json`
x_coord	Number	1	Location for this node.
y_coord	Number	1	Location for this node.
z_coord	Number	1	Location for this node.
node	Node	1	A link to he node currently plugged to this socket.

Node [application/ctf.Node+json]

The Node describes properties (platform, interfaces, sensors, actuators, radio technology, mobility) and state (power, coordinates) of a testbed node.

Node resource model contains the following fields:

Field Name	Type	Occurs	Description
uri	URI	1	A GET against this URI refreshes the representation of this resource.
name	String	1	A given name for this resource.
media_type	String	1	`application/ctf.Node+json`
platform	Platform	1	The Platform represent the hardware characteristics of this node.
interfaces	Interface[]	1	Interfaces this node supports.
sensors	Sensor[]	0..1	A list of sensors featured by this Node.
actuators	Actuator[]	0..1	A list of actuators featured by this Node.
radio	Radio[]	0..1	The radio technology adopted by this node.
mobility	Mobility	0..1	If present indicates the kind of mobility featured by this socket.
power	Boolean	1	If *true* means power-on, if *false* power off.
image	Image	1	The Image file that is to be loaded in this node.
x_coord	Number	1	Location for this node (inherited by the socket this node is plugged to)
y_coord	Number	1	Location for this node (inherited by the socket this node is plugged to)
z_coord	Number	1	Location for this node (inherited by the socket this node is plugged to)

Platform [application/ctf.Platform+json]

The Platform resource represents a set of hardware components of a Node.

Platform resource model contains the following fields:

Field Name	Type	Occurs	Description
uri	URI	1	A GET against this URI refreshes the representation of this resource.
name	String	1	A given name for this resource.
media_type	String	1	`application/ctf.Platform+json`
cpu	String	1	An identifier representing the model of CPU featured by this platform.
memory	Number	1	The size of the memory featured by this platform.
image_formats	ImageFormat[]	1	A list of supported image formats for this platform.
bandwidth_lower_bound	Integer	1	The lower bound in MHz for this platform (used to calculate the collision domain).
bandwidth_upper_bound	Integer	1	The upper bound in MHz for this platform (used to calculate the collision domain).

Interface [application/ctf.Interface+json]

The Interface defines a communication interface supported by nodes for communication between the testbed infrastructure and the SUT. Since different kinds of interface may be supported by nodes it is important to have this resource universally defined within the CTF so that every member testbed may refer to this resource in the description of a given node.

Interface resource model contains the following fields:

Field Name	Type	Occurs	Description
uri	URI	1	A GET against this URI refreshes the representation of this resource.
name	String	1	A given name for this resource.
media_type	String	1	`application/ctf.Interface+json`

Radio [application/ctf.Radio+json]

The Radio resource standardizes radio technologies adopted in the CTF. Since different kinds of radio technologies may be supported by nodes it is important to have this resource universally defined within the CTF so that every member testbed may refer to this resource in the description of a given node.

Radio resource model contains the following fields:

Field Name	Type	Occurs	Description
uri	URI	1	A GET against this URI refreshes the representation of this resource.
name	String	1	A given name for this resource.
media_type	String	1	`application/ctf.Radio+json`

Sensor [application/ctf.Sensor+json]

The Sensor resource represents a device dedicated to measuring a particular physical parameter like temperature, pressure, humidity, light or sound. Since different kinds of sensor may be supported by nodes it is important to have this resource universally defined within the CTF so that every member testbed may refer to this resource in the description of a given node.

Sensor resource model contains the following fields:

Field Name	Type	Occurs	Description
uri	URI	1	A GET against this URI refreshes the representation of this resource.
name	String	1	A given name for this resource.
media_type	String	1	`application/ctf.Sensor+json]`
description	String	1	A human-readable description of this Sensor.
values	Number[]	1	The measured values of this Sensor.
units	String[]	1	The units of measurement by which values are expressed.

Actuator [application/ctf.Actuator+json]

The Actuator resource represents a device which can operate a particular action in the physical world, i.e. sound speakers, LED or robots. Since different kinds of actuator may be supported by nodes it is important to have this resource universally defined within the CTF so that every member testbed may refer to this resource in the description of a given node.

Actuator resource model contains the following fields:

Field Name	Type	Occurs	Description
uri	URI	1	A GET against this URI refreshes the representation of this resource.
name	String	1	A given name for this resource.
media_type	String	1	`application/ctf.Actuator+json`
description	String	1	A human-readable description of this Actuator.
values	Number[]	1	The values this Actuator should assume.
units	String[]	1	The units of measurement by which values are expressed.

Mobility [application/ctf.Mobility+json]

The Mobility resource defines a mobility feature for a node. Since different kinds of mobility may be supported by nodes it is important to have this resource universally defined within the CTF so that every member testbed may refer to this resource in the description of a given node. For example nodes may move either in a 2-dimensional or 3-dimensional space.

Mobility resource model contains the following fields:

Field Name	Type	Occurs	Description
uri	URI	1	A GET against this URI refreshes the representation of this resource.
name	String	1	A given name for this resource.
media_type	String	1	`application/ctf.Mobility+json`
description	String	1	A description for this resource.

ImageFormat [application/ctf.ImageFormat+json]

The ImageFormat resource defines a format for the Image file that can be burned on a given node. Since different platforms may support different image formats it is important to have this resource universally defined within the CTF.

ImageFormat resource model contains the following fields:

Field Name	Type	Occurs	Description
uri	URI	1	A GET against this URI refreshes the representation of this resource.
name	String	1	A given name for this resource.
media_type	String	1	`application/ctf.ImageFormat+json]`

3.3 Requests and Responses

In the following tables we present a description for the application of the four HTTP methods to the resources introduced in the previous section. In the *Status Codes* column, the following codes 400, 401, 403, 404, 406 and 500 are intended to be always implicit since they may refer to any kind of HTTP Request with the same meaning, while we specify only those HTTP Status Codes that have a different meaning according to the different HTTP Methods.

3.3.1 Testbed Federation API

Project [application/ctf.Project+json]

Method	Description	Status Codes
GET	Returns the JSON representation of the Project resource.	200, 204, 304, 410
PUT	Modifying JSON representation for the Project including Users, Experiments, Testbeds or Jobs.	200, 202, 409, 412
POST	Method not allowed.	405
DELETE	Delete the resource specified by the URI	200, 409, 412

Experiment [application/ctf.Experiment+json]

Method	Description	Status Codes
GET	Returns the JSON representation of the Experiment resource.	200, 204, 304, 410
PUT	Modifying JSON representation for the Experiment including PropertySets, VirtualNodes, VirtualTasks or Images.	200, 202, 409, 412
POST	Method not allowed.	405
DELETE	Delete the resource specified by the URI	200, 409, 412

PropertySet [application/ctf.PropertySet+json]

Method	Description	Status Codes
GET	Returns the JSON representation of the PropertySet resource.	200, 204, 304, 410
PUT	Modifying JSON representation for the PropertySet including User, Experiment, Platforms, Radios, Interfaces, Sensors, Actuators, Mobility and node number.	200, 202, 409, 412
POST	Method not allowed.	405
DELETE	Delete the resource specified by the URI	200, 409, 412

VirtualNodeGroup [application/ctf.VirtualNodeGroup+json]

Method	Description	Status Codes
GET	Returns the JSON representation of the VirtualNodeGroup resource.	200, 204, 304, 410
PUT	Modifying JSON representation for the VirtualNodeGroup including references to VirtualNodes contained.	200, 202, 409, 412
POST	Method not allowed.	405
DELETE	Delete the resource specified by the URI	200, 409, 412

VirtualNode [application/ctf.VirtualNode+json]

Method	Description	Status Codes
GET	Returns the JSON representation of the VirtualNode resource.	200, 204, 304, 410
PUT	Method not allowed.	405
POST	Method not allowed.	405
DELETE	Delete the resource specified by the URI	200, 409, 412

VirtualTask [application/ctf.VirtualTask+json]

Method	Description	Status Codes
GET	Returns the JSON representation of the VirtualTask resource.	200, 204, 304, 410
PUT	Modifying JSON representation for the VirtualTask including values of method, headers, payload or target fields.	200, 202, 409, 412
POST	Method not allowed.	405
DELETE	Delete the resource specified by the URI	200, 409, 412

3.3.2 TA API and TF API

NodeGroup [application/ctf.NodeGroup+json]

Method	Description	Status Codes
GET	Returns the JSON representation of the NodeGroup resource.	200, 204, 304, 410
PUT	Modifying JSON representation for the NodeGroup including references to Nodes contained.	200, 202, 409, 412
POST	Method not allowed.	405
DELETE	Delete the resource specified by the URI	200, 409, 412

Image [application/ctf.Image+json]

Method	Description	Status Codes
GET	Returns the JSON representation of the Image resource.	200, 204, 304, 410
PUT	Modifying JSON representation for the Image updating mapping_keys, mapping_values, binary or image format resource reference.	200, 202, 409, 412
POST	Method not allowed.	405
DELETE	Delete the resource specified by the URI	200, 409, 412

Task [application/ctf.Task+json]

Method	Description	Status Codes
GET	Returns the JSON representation of the Task resource.	200, 204, 304, 410
PUT	Modifying JSON representation for the Task, should only affect the Status resource reference.	200, 202, 409, 412
POST	Method not allowed.	405
DELETE	Delete the resource specified by the URI	200, 409, 412

Trace [application/ctf.Trace+json]

Method	Description	Status Codes
GET	Returns the JSON representation of the Trace resource.	200, 204, 304, 410
POST	Method not allowed.	405
POST	Method not allowed.	405
DELETE	Delete the resource specified by the URI	200, 409, 412

Log [application/ctf.Log+json]

Method	Description	Status Codes
GET	Returns the JSON representation of the Log resource.	200, 204, 304, 410
POST	Method not allowed.	405
POST	Method not allowed.	405
DELETE	Delete the resource specified by the URI	200, 409, 412

Socket [application/ctf.Socket+json]

Method	Description	Status Codes
GET	Returns the JSON representation of the Socket resource.	200, 204, 304, 410
PUT	Upload a new JSON representation for a given Socket.	200, 202, 409, 412
POST	Method not allowed.	405
DELETE	Delete the resource specified by the URI	200, 409, 412

Node [application/ctf.Node+json]

Method	Description	Status Codes
GET	Returns the JSON representation of the Node resource.	200, 204, 304, 410
PUT	Upload a new JSON representation for a given Node.	200, 202, 409, 412
POST	Method not allowed.	405
DELETE	Method not allowed.	405

Platform [application/ctf.Platform+json]

Method	Description	Status Codes
GET	Returns the JSON representation of the Platform resource.	200, 204, 304, 410
PUT	Method not allowed.	405
POST	Method not allowed.	405
DELETE	Method not allowed.	405

Interface [application/ctf.Interface+json]

Method	Description	Status Codes
GET	Returns the JSON representation of the Interface resource.	200, 204, 304, 410
PUT	Method not allowed.	405
POST	Method not allowed.	405
DELETE	Method not allowed.	405

Radio [application/ctf.Radio+json]

Method	Description	Status Codes
GET	Returns the JSON representation of the Radio resource.	200, 204, 304, 410
PUT	Method not allowed.	405
POST	Method not allowed.	405
DELETE	Method not allowed.	405

Sensor [application/ctf.Sensor+json]

Method	Description	Status Codes
GET	Returns the JSON representation of the Sensor resource.	200, 204, 304, 410
PUT	Method not allowed.	405
POST	Method not allowed.	405
DELETE	Method not allowed.	405

Actuator [application/ctf.Actuator+json]

Method	Description	Status Codes
GET	Returns the JSON representation of the Actuator resource.	200, 204, 304, 410
PUT	Method not allowed.	405
POST	Method not allowed.	405
DELETE	Method not allowed.	405

Mobility [application/ctf.Mobility+json]

Method	Description	Status Codes
GET	Returns the JSON representation of the Mobility resource.	200, 204, 304, 410
PUT	Method not allowed.	405
POST	Method not allowed.	405
DELETE	Method not allowed.	405

ImageFormat [application/ctf.ImageFormat+json]

Method	Description	Status Codes
GET	Returns the JSON representation of the ImageFormat resource.	200, 204, 304, 410
PUT	Method not allowed.	405
POST	Method not allowed.	405
DELETE	Method not allowed.	405

3.4 Invocation Examples

3.4.1 User creates a new Project

```
POST /projects/
Host: federation.com
Authorization: Basic XXXXXXXXXX
Accept: application/ctf.Project+json
Content-Length: nnnn
Content-Type: application/ctf.Project+json

{
    "name" : "Test Projest",
    "description" : "A simple test project",
    "users" :
    [
        {
            "uri" : "http://federation.com/users/123",
            "name" : "John Smith",
            "media_type" : "application/ctf.User+json"
        }
    ]
}

HTTP/1.1 201 Created
Location: http://federation.com/projects/456
```

3.4.2 User creates a new Experiment

```
POST /projects/456/experiments
Host: federation.com
Authorization: Basic XXXXXXXXXX
Accept: application/ctf.Experiment+json
Content-Length: nnnn
Content-Type: application/ctf.Experiment+json

{
    "name" : "Test Experiment",
    "description" : "A simple test experiment",
    "user" :
    {
        "uri" : "http://federation.com/users/123",
        "name" : "John Smith",
        "media_type" : "application/ctf.User+json"
    },
    "project" :
    {
        "uri" : "http://federation.com/projects/456",
        "name" : "Test Projest",
        "media_type" : "application/ctf.Project+json"
    },
```

```
    "sharing" : "public"
}

HTTP/1.1 201 Created
Location: http://federation.com/projects/456/experiments/789
```

3.4.3 User creates two PropertySets

```
POST /projects/456/experiments/789/property_sets
Host: federation.com
Authorization: Basic XXXXXXXXXX
Accept: application/ctf.PropertySet+json
Content-Length: nnnn
Content-Type: application/ctf.PropertySet+json

{
    "name" : "Test Property Set 1",
    "description" : "Data collection property set",
    "node_count" : "10",
    "platform_type" :
    {
        "uri" : "http://federation.com/platforms/165",
        "name" : "TelosB",
        "media_type" : "application/ctf.Platform+json"
    },
    "interfaces" :
    [
        {
            "uri" : "http://federation.com/interfaces/769",
            "name" : "USB",
            "media_type" : "application/ctf.Interface+json"
        }
    ],
    "sensors" :
    [
        {
            "uri" : "http://federation.com/sensors/587",
            "name" : "Temperature",
            "media_type" : "application/ctf.Sensors+json"
        },
        {
            "uri" : "http://federation.com/sensors/639",
            "name" : "Light",
            "media_type" : "application/ctf.Sensors+json"
        }
    ],
    "experiment" :
    {
        "uri" : "http://federation.com/projects/456/experiments/789",
        "name" : "Test Experiment",
        "media_type" : "application/ctf.Experiment+json"
```

```
    }
}

HTTP/1.1 201 Created
Location: http://federation.com/projects/456/experiments/789/property_sets/659

POST /projects/456/experiments/789/property_sets
Host: federation.com
Authorization: Basic XXXXXXXXXX
Accept: application/ctf.PropertySet+json
Content-Length: nnnn
Content-Type: application/ctf.PropertySet+json

{
    "name" : "Test Property Set 2",
    "description" : "Data sink",
    "node_count" : "1",
    "platform_type" :
    {
        "uri" : "http://federation.com/platforms/165",
        "name" : "TelosB",
        "media_type" : "application/ctf.Platform+json"
    },
    "interfaces" :
    [
        {
            "uri" : "http://federation.com/interfaces/769",
            "name" : "USB",
            "media_type" : "application/ctf.Interface+json"
        }
    ],
    "mobility" :
    {
        "uri" : "http://federation.com/mobilities/295",
        "name" : "2D",
        "media_type" : "application/ctf.Mobility+json"
    }
    "experiment" :
    {
        "uri" : "http://federation.com/projects/456/experiments/789",
        "name" : "Test Experiment",
        "media_type" : "application/ctf.Experiment+json"
    }
}

HTTP/1.1 201 Created
Location: http://federation.com/projects/456/experiments/789/property_sets/825
```

3.4.4 User uploads two Image files

```
POST /projects/456/experiments/789/images
```

```
Host: federation.com
Authorization: Basic XXXXXXXXXX
Accept: application/ctf.Image+json
Content-Length: nnnn
Content-Type: application/ctf.Image+json

{
    "name" : "Test Image 1",
    "description" : "Image for data collection",
    "image_format" :
    {
        "uri" : "http://federation.com/image_formats/658",
        "name" : "IHEX",
        "media_type" : "application/ctf.ImageFormat+json"
    },
    "content" : "YYYYYYYYYYYYYYYYYYYYYYYYYYYYYY",
    "mapping_keys" : ["NODE_ID"],
    "mapping_values" : ["http://federation.com/testbeds/{testbed}/nodes/{node}/id"]
}

HTTP/1.1 201 Created
Location: http://federation.com/projects/456/experiments/789/images/648

POST /projects/456/experiments/789/images
Host: federation.com
Authorization: Basic XXXXXXXXXX
Accept: application/ctf.Image+json
Content-Length: nnnn
Content-Type: application/ctf.Image+json

{
    "name" : "Test Image 2",
    "description" : "Image for data sink",
    "image_format" :
    {
        "uri" : "http://federation.com/image_formats/658",
        "name" : "IHEX",
        "media_type" : "application/ctf.ImageFormat+json"
    },
    "content" : "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX",
    "mapping_keys" : ["NODE_ID"],
    "mapping_values" : ["http://federation.com/testbeds/{testbed}/nodes/{node}/id"]
}

HTTP/1.1 201 Created
Location: http://federation.com/projects/456/experiments/789/images/346
```

3.4.5 User creates two Virtual Tasks

```
POST /projects/456/experiments/789/virtual_tasks
Host: federation.com
```

```
Authorization: Basic XXXXXXXXXX
Accept: application/ctf.VirtualTask+json
Content-Length: nnnn
Content-Type: application/ctf.VirtualTask+json

{
    "name" : "Test Virtual Task",
    "description" : "Burning Image 1 on Virtual NodeGroup 1",
    "eta" : "0",
    "method" : "PUT",
    "headers" : "
        Host: federation.com
        Authorization: Basic XXXXXXXXXX
        Accept: application/ctf.VirtualNodeGroup+json
        Content-Length: nnnn
        Content-Type: application/ctf.VirtualNodeGroup+json
    "
    "payload" : "
        "image" :
        {
            "uri" : "http://federation.com/projects/456/experiments/789/images/648",
            "name" : "Test Image 1",
            "media_type" : "application/ctf.Image+json"
        },
        "eta" : "0"
    "
    "target" : "/projects/456/experiments/789/virtual_node_groups/659"
}

HTTP/1.1 201 Created
Location: http://federation.com/projects/456/experiments/789/virtual_tasks/729

POST /projects/456/experiments/789/virtual_tasks
Host: federation.com
Authorization: Basic XXXXXXXXXX
Accept: application/ctf.VirtualTask+json
Content-Length: nnnn
Content-Type: application/ctf.VirtualTask+json

{
    "name" : "Test Virtual Task",
    "description" : "Burning Image 2 on Virtual NodeGroup 2",
    "eta" : "0",
    "method" : "PUT",
    "headers" : "
        Host: federation.com
        Authorization: Basic XXXXXXXXXX
        Accept: application/ctf.VirtualNodeGroup+json
        Content-Length: nnnn
        Content-Type: application/ctf.VirtualNodeGroup+json
    ",
```

```
    "payload" : "
        "image" :
        {
            "uri" : "http://federation.com/projects/456/experiments/789/images/346",
            "name" : "Test Image 2",
            "media_type" : "application/ctf.Image+json"
        }
    ",
    "target" : "/projects/456/experiments/789/virtual_node_groups/825"
}

HTTP/1.1 201 Created
Location: http://federation.com/projects/456/experiments/789/virtual_tasks/628
```

3.4.6 User explores the federation by submitting the experiment definition

```
GET /testbeds?experiment=http://federation.com/projects/456/experiments/789
Host: federation.com
Authorization: Basic XXXXXXXXXX
Accept: application/ctf.Testbed+json

HTTP/1.1 200 OK
Content-Length: nnnn
Content-Type: application/ctf.Testbed+json

[
    {
        "uri" : "http://federation.com/testbeds/234",
        "name" : "Testbed A",
        "media_type" : "application/ctf.Testbed+json"
    },
    {
        "uri" : "http://federation.com/testbeds/872",
        "name" : "Testbed B",
        "media_type" : "application/ctf.Testbed+json"
    },
    {
        "uri" : "http://federation.com/testbeds/926",
        "name" : "Testbed C",
        "media_type" : "application/ctf.Testbed+json"
    },
]
```

3.4.7 User creates a new Job for a given Experiment

```
POST /projects/456/experiments/789/jobs
Host: federation.com
Authorization: Basic XXXXXXXXXX
Accept: application/ctf.Job+json
Content-Length: nnnn
Content-Type: application/ctf.Job+json
```

```
{
    "name" : "Test Job",
    "description" : "A simple test job",
    "owner" :
    {
        "uri" : "http://federation.com/users/235",
        "name" : "John Smith",
        "media_type" : "application/ctf.User+json"
    },
    "project" :
    {
        "uri" : "http://federation.com/projects/623",
        "name" : "My sample project",
        "media_type" : "application/ctf.Project+json"
    },
    "experiment" :
    {
        "uri" : "http://federation.com/projects/623/experiments/812",
        "name" : "My sample experiments",
        "media_type" : "application/ctf.Experiment+json"
    },
    "testbed" :
    {
        "uri" : "http://federation.com/testbeds/234",
        "name" : "Testbed A",
        "media_type" : "application/ctf.Testbed+json"
    },
    "platform" :
    {
        "uri" : "http://federation.com/platforms/653",
        "name" : "TelosB",
        "media_type" : "application/ctf.Platform+json"
    },
    "datetime_from" : "2009-11-18 14:00:00",
    "datetime_to" : "2009-11-16 18:00:00",

}

HTTP/1.1 201 Created
Location: http://federation.com/projects/456/experiments/789/jobs/124
```

Bibliography

[1] T. Berners-Lee, R. Fielding, and L. Masinter. Uniform resource identifiers (uri): Generic syntax, 2005.

[2] D. Crockford. The application/json media type for javascript object notation (json), 2006.

[3] L. Daigle, D.W. Van Gulik, R. Iannella, and P. Faltstrom. Uniform resource names (urn) namespace definition mechanisms, 2002.

[4] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – http/1.1, 1999.

[5] R.T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

[6] N. Freed and Borenstein N. Multipurpose internet mail extensions (mime) part two: Media types, 1996.

[7] Omprakash Gnawali, Rodrigo Fonseca, Kyle Jamieson, David Moss, and Philip Levis. Collection tree protocol. Technical Report SING-09-01, Stanford University, 2009.

[8] Federated Login for Google Account Users. `http://code.google.com/apis/accounts/docs/OpenID.html`. [Online; accessed: 02.10.2009].

[9] IANA. `http://www.iana.org/assignments/urn-namespaces/`. [Online; accessed: 02.10.2009].

[10] Sun Cloud API. `http://kenai.com/projects/suncloudapis/`. [Online; accessed: 02.10.2009].

[11] Kerberos Authentication Protocol. `http://web.mit.edu/Kerberos/`. [Online; accessed: 02.10.2009].

[12] OAuth. `http://oauth.net/`. [Online; accessed: 02.10.2009].

[13] OpenID. `http://openid.net/`. [Online; accessed: 02.10.2009].

[14] ProtoGENI. `http://www.protogeni.net/`. [Online; accessed: 02.10.2009].

[15] Shibboleth (R). `http://shibboleth.internet2.edu/`. [Online; accessed: 02.10.2009].

[16] Wisebed. http://www.wisebed.eu/. [Online; accessed: 02.10.2009].