Division of Informatics
The DICE Project
Node Profile Specification

Release 1-1.05 Tue Jul 24 09:32:12 2001

Release Notes

This document supersedes and obsoletes all previous releases of the node profile specification document.

Table of Contents

1 About This Document

This is the main body of the text. You really ought to try and read this!

A side-bar note is to used to elaborate on a definition like a glossary entry; to give a list of example values; to provide a footnote like aside to the main flow of text. Side-bar notes are generally horizontally aligned with the paragraph they are associated with.

> • Boxes like this raise important or interesting points that are supplemental to the main flow of the rest of the text or to group together items under a certain heading.

The `NAME` font is used for the names of all objects in the description.

The *italic* and **bold** fonts are used for values or just to emphasize a particular word or phrase.

> • Boxes like this group information on the same subject.

2 Node Profile

A node profile is a per-host configuration description. It consists of sets of key/value pairs with support for representing complex structures such as lists, records, trees and large embedded or referenced objects. For development and prototyping the existing LCFG NIS map is expanded and converted into a node profile for each host via an adaptor.

A node profile is represented in XML and has a corresponding SGML DTD for validation. It looks much like a simplified Strawman RDF syntax. The XML-Namespace syntax is used to support separate name groups within a profile. The URI notation and a pseudo XPath notation are used for references and element addressing.

Some rudimentary knowledge of the following recommendations and specifications is useful for understanding this specification.

- XML and DTD ("http://www.w3.org/TR/2000/REC-xml-20001006")
- XML-Namespace ("http://www.w3.org/TR/1999/REC-xml-names-19990114/")
- RDF ("http://www.w3.org/TR/REC-rdf-syntax/")
- RDF-Schema ("http://www.w3.org/TR/2000/CR-rdf-schema-20000327/")
- Strawman ("http://www.w3.org/DesignIssues/Syntax")
- XPath ("http://www.w3.org/TR/xpath.html")
- URI ("http://www.ietf.org/rfc/rfc2396.txt")

Some additional documents of ancillary interest are listed below.

- XPointer ("http://www.w3.org/TR/xptr")
- DSML ("http://www.dsml.org/")
- XML-Include ("http://www.w3.org/TR/xinclude/")
- XML-Schema ("http://www.w3.org/TR/xmlschema-0/", "http://www.w3.org/TR/xmlschema-1/" and "http://www.w3.org/TR/xmlschema-2/")
- XSL ("http://www.w3.org/TR/xsl/")
- XSLT ("http://www.w3.org/TR/xslt.html")

2-1 Structure

The structure is very like a filesystem: a resource is a directory, a property is a file, a value is the contents of a file.

An XML representation is used to hold resources and property/value pairs for resources.

Below is an example of a basic profile container, the actual content of which is expanded on in later sections.

Each distribution web server could rewrite the schema URI references to point at copies held locally.

```
<?xml version="1.0"?>
<!DOCTYPE profile SYSTEM
 "http://cfg.inf.ed.ac.uk/1.0/profile.dtd">
<profile
 xmlns="http://cfg.inf.ed.ac.uk/1.0/profilens"
 xmlns:cfg="http://cfg.inf.ed.ac.uk/1.0/cfgns">

 <!-- RESOURCES -->

</profile>
```

The **profile** element (root node) contains zero or more distinct resource elements. The permissible resource elements are specified in the DTD.

The ordering of elements and child elements in the profile itself is preserved through to the API and is significant.

2-1-1 Resources and Properties

A resource element either contains multiple and distinct property and resource elements or contains multiple instances of the same resource element or is empty. The permissible content of a resource element is specified in the DTD.

A property element is either empty or contains a literal value. The permissible property elements and values in each resource element are specified in the DTD.

Any literal value must conform to XML special symbol escaping notation, so an embedded less than symbol (<) must be written as **<** for example.

```
<vmware>
 <encrypt>no</encrypt>
 <license>118456</license>
</vmware>
```

In the example above the **vmware** resource contains two properties, **encrypt** and **license**.

An element can be either a container of other elements (a resource) or have a literal value (a property), it can never be both. This restriction is specified in the DTD. A configuration component with more than one value must either be represented as some kind of embedded list in a property value or as a nested resource.

Multiple instances of the same resource or property element can occur intermixed with other elements only when all instances of the same

element are distinguished by use of the **cfg:context** attribute (a requirement that is specified in the DTD). This is used for context switching and as far as the client node is concerned only one instance of the element is ever visible through the API, the particular instance is determined by the client node cache manager and the current context.

2-1-2 Nested Resources

Nested resource elements allow a tree like structure of configuration components to be built to any level.

```
<disk>
 <device>/dev/hda</device>
 <partitions>
   <partition>
     <size>1000</size>
     <mount>/</mount>
   </partition>
   <partition>
     <size>250</size>
     <mount>/tmp</mount>
   </partition>
 </partitions>
</disk>
```

In the example above the **disk** resource consists of a **device** property and a **partitions** resource. The **partitions** resource contains multiple instances of the **partition** resource. Each **partition** resource contains a **size** and **mount** property.

2-1-3 Named Elements

Multiple instances of the same resource or property element can optionally be given a name so that they can be addressed by name rather than by index. This is specified with the **cfg:name** attribute.

```
<disk>
 <device>/dev/hda</device>
 <partitions>
   <partition cfg:name="root">
     <size>1000</size>
     <mount>/</mount>
   </partition>
   <partition cfg:name="tmp">
     <size>250</size>
     <mount>/tmp</mount>
   </partition>
 </partitions>
</disk>
```

Multiple instances of the same element must either be all named or all unnamed, mixes of named and unnamed elements of the same type are not permitted. This restriction is specified in the DTD. The DTD also restricts the use of the **cfg:name** attribute to elements permitted to appear more than once.

The name does not need to be unique in the profile but must be unique against all siblings of the same element type. The DTD cannot validate this (without the specific name values being encoded into it).

Without explicit names the names of multiple instances of elements are a constructed index position. The names of single instances of elements are the same as the element type.

2-1-4 Encoding LCFG

For development and prototyping, the existing LCFG key/value pairs will be mapped into resources and properties in the profile using the new syntax structure as above but under a special resource tree and with a separate namespace. This will allow them to be slowly migrated into the new structure.

```
<profile>
 .
 .
 .
 <lcfg xmlns='http://cfg.inf.ed.ac.uk/1.0/lcfgns'>
   <vmware>
     <encrypt>yes</encrypt>
   </vmware>
   .
   <update>
     .
     .
     .
   </update>
 </lcfg>
</profile>
```

2-2 Addressing

The client node profiles are held and distributed via one or more web servers. The whole profile for a client node is fetched as a single unit except for the values of property elements which are references rather than actually embedded in the profile.

The server path is passed to the client very early in the boot sequence via DHCP.

Authentication and authorization for a client to access its profile is handled either transparently by HTTP (e.g. using SSL certificates) or inside HTTP (e.g. credentials passed as part of the HTTP request). Communication over the network is encrypted (e.g. SSL also).

In order to address individual elements (or attributes) in the XML representation of the configuration profile the client node API supports a simple directory path access notation.

2-2-1 Base Address

The profile for a client node is handed back from a web server either from a various URI addresses or via a CGI script interface. This allows the profile to be requested in different ways based on the keys available at the time.

```
http://cfg.inf.ed.ac.uk/profile?hn=fqhname
```

In the example above *fqhname* is the fully qualified host name of the client node (for example *wibble.dcs.ed.ac.uk*). The above returns *profile.xml* containing the full XML profile for that client node as described in the previous section. Other possible lookups could be by host id, IP address or MAC address for example. The lookup may also need to include suitable credentials if SSL over HTTP is not implemented.

A fully qualified hostname allows profiles for client nodes in multiple domains to be hosted from the same configuration server.

Additional addresses could be supported for distributing information common to all client nodes. These could hold the profile DTD as well as any additional files which are used by all nodes, such as an RPM repository for example.

A per-node directory may also be supported to contain additional files specific to each node which are part of that nodes profile but are not held in the profile itself (referenced large objects for example).

2-2-2 Resource and Property Address

The client node fetches a profile for itself using an appropriate address as above. Addressing elements within that profile is then achieved through the client node API using a simple directory / file path notation. The traversal through a node profile resembles very much a directory tree structure traversal.

```
/vmware
```

The above example addresses the **vmware** resource element. The returned value is a list of element names contained by the resource.

```
/vmware/encrypt
```

The above example addresses the **encrypt** property element of the **vmware** resource element. The returned value is the literal value of the **encrypt** property.

Multiple elements are addressed in the same way.

```
/disk/partitions
```

The above example addresses the **partitions** resource element of the **disk** resource element. Since the **partitions** resource contains multiple instances of the same resource the returned value is either a list of the names (if defined with **cfg:name**) or a list of generated names (index values). In the former case individual elements can be addressed as below:

```
/disk/partitions/root
/disk/partitions/tmp
```

Whereas in the latter case individual elements are addressed as below:

```
/disk/partitions/1
/disk/partitions/2
```

2-2-3 Element Naming

Resource and property element names (as well as names given as a value for the **cfg:name** attribute) should conform to a more restricted form of the XML name token which excludes the extended and combining character sets as well as the colon character (since XML-Namespace is used).

Hence names must begin with an alphabetic character or an underscore character and can only include alphanumeric characters and the underscore character.

2-3 Namespaces

The default namespace is the profile namespace and there are no reserved words so resource and property elements can be named anything. Elements and attributes from other namespaces are fully qualified. Generally just profile attributes need qualification. Below is a list of the basic namespaces.

- PRF (profile resource and property elements, default namespace)
- CFG (node profile attributes)
- LCFG (special namespace for holding old LCFG resource and property element names)

A DTD does not support the same element type having different definitions based on parent container. In these situations additional custom profile namespaces may need to be specified, probably on a per-resource basis, to prevent element names in one resource clashing with element names in another resource.

```
<vmware
 xmlns="http://cfg.inf.ed.ac.uk/1.0/profilens/vmwarens">
 <encrypt>yes</encrypt>
</vmware>
<samba
 xmlns="http://cfg.inf.ed.ac.uk/1.0/profilens/sambans">
 <encrypt>
    <method>3DES</method>
    <enable>yes</enable>
 </encrypt>
</samba>
```

Since a DTD does not support XML-Namespace the above could not be easily validated.

2-4 Values

The value of a property as a literal is always parsed character data, a restriction specified in the DTD. The interpretation of a value is by default as a character string but other interpretations can be specified using the **cfg:type** attribute.

```
<named>
 <server>x.y.z</server>
 <serial cfg:type="integer">11566778</serial>
 <enable cfg:type="boolean">yes</enable>
</named>
```

2-4-1 Value Formats

The format of a literal value depends on its type. The accepted types and formats are listed below.

An **INT** or **LONG** literal is a sequence of digits optionally preceeded by a sign (plus or minus). It is considered to be an octal number if the sequence begins with a zero digit. An octal number cannot include the digits **8** or **9**. Alternatively if the sequence of digits starts with **0x** or **0X** it is considered to be a hexadecimal number and can include **a** or **A** through **f** or **F** as digits representing values 10 through 15 respectively. An **INT** has a range of -2^{31} through $2^{31}-1$. A **LONG** has a range of -2^{63} through $2^{63}-1$. Values outside these ranges are illegal.

A **FLOAT** or **DOUBLE** literal has an integer part, a decimal point, a fractional part, an **e** or **E** and an optional integer exponent. The integer part, the fractional part and the optional exponent are a sequence of digits. The integer part and the exponent can optionally be preceeded by a sign (plus or minus). The integer part or the fractional part (but not both) can be missing and the decimal point or the exponent (but not both) can be missing. A **FLOAT** or **DOUBLE** has a range of -1.0e+38 to +1.0e+38 with a precision of 7 digits or 16 digits respectively. Values outside these ranges are illegal.

A **BOOLEAN** literal can be **yes**, **true**, **on** or **1** (one digit) to mean true and **no**, **false**, **off** or **0** (zero digit) to mean false. Any other value is illegal.

A **STRING** literal can be any parsed character data.

2-4-2 Objects

External linked objects or embedded objects can also be specified as the value of a property element. This is achieved using some special case values of the **cfg:type** attribute which define the value as being an object (rather than a literal) and how to access the object.

```
<named>
 <boot_template cfg:type="fetch">
   http://cfg.ed.ac.uk/host/xyz.dcs.ed.ac.uk/named.boot
 </boot_template>
</named>
```

In the example above the value of the **boot_template** property is the content of the file referenced by the URI.

Objects can be embedded as in the example below.

```
<named>
 <boot_template cfg:type="embed" cfg:encoding=base64">
   <!-- B64 encoded string of object -->
 </boot_template>
</named>
```

Either linked or embedded objects defined as above are returned directly as the value of the property, no further parsing is carried out.

For example, an embedded object could contain XML which would be passed directly to the component method as the value for that property, the component method would need to be able to parse XML to interpret the value.

Very large linked objects can be streamed directly through the client node API (using a local named pipe, the client daemon passing data into one end which is consumed by the component at the other end) by specifying **stream** as a value for the **cfg:type** attribute.

2-5 Attributes

Attributes are used by the client node but need not necessarily be visible through the client node API.

Global profile attributes are used to define meta-data on resources and properties. Attributes are used to define timestamps, access control, dynamic values and context switching. They are also used for naming and typing properties as shown in earlier sections.

Profile attributes are in a separate namespace so they don't have to be treated as reserved words in the default namespace. New ones can then be added without having to potentially alter existing profiles.

Profile attributes are defined in the configuration namespace and must always be prefixed with **cfg:**. All profile attributes are optional. Most can be used with both resource and property elements but some can only be used with one type of element which is specified in the DTD.

There are no ordering constraints on attributes and no significance should be interpreted from a specific ordering.

2-5-1 Name

The name of a resource or property element is used for addressing and distinguishing multiple instances of the same element. The name of an element can be defined using the **cfg:name** attribute. By default for single instance elements the name is the same as the element type. By default for multiple instance elements the name is the index position number of the element. An element can only be addressed by its specified name or by its generated index number, never by both.

```
<vmware>
 <licences>
    <license cfg:name="user">116778</license>
    <license cfg:name="system">117886</license>
 </licenses>
 <encrypt>yes</encrypt>
 <images>
    <image>basic</image>
    <image>advanced</image>
    <image>custom</image>
 </images>
</vmware>
```

In the example above the following addresses are valid.

```
PATH                        VALUE
/vmware                     licenses; encrypt; images
/vmware/licenses            user; system
/vmware/licenses/user       "116778"
/vmware/licenses/system     "117886"
/vmware/encrypt             "yes"
/vmware/images              1; 2; 3
/vmware/images/1            "basic"
/vmware/images/2            "advanced"
/vmware/images/3            "custom"
```

2-5-2 Type

The type of an element. The type of an element is defined using the **cfg:type** attribute. The type of a resource element is always **RESOURCE**, although this can also be made explicit with this attribute (to ensure correct identification of an empty element for example when there is no DTD to refer to). A resource element cannot be any other type and this restriction is specified in the DTD. The type of a property element determines the interpretation of its literal value. The type of a property element defaults to **STRING**, as specified in the DTD. The full set of permissible types is specified in the DTD.

Available types for property elements are:

- String
- Boolean
- Int
- Long
- Float
- Double
- Stream
- Fetch (BLOB)
- Embed (BLOB)

Available types for resource elements are:

- Resource

In addition there are three special types for a property element.

The **FETCH** type indicates that the value of the property is a URI the contents of which are fetched by the client node and returned as the value of the property. The **STREAM** type indicates that the value of the property is a URI the contents of which are streamed directly through the client node API.

```
<vmware>
 <config_template_file cfg:type="fetch">
    http://cfg.inf.ed.ac.uk/etc/vmware.conf
 </config_template_file>
 <image cfg:type="stream">
    http://cfg.inf.ed.ac.uk/rpms/vmware.img.1-34.rpm
 </image>
</vmware>
```

The **EMBED** type indicates that the value of the property contains an inlined (and possibly encoded object) which the client node returns as the value of the property. The decoding of an embedded object is handled by the client language library.

```
<vmware>
 <image cfg:type="embed" cfg:encoding="base64">
    <!-- B64 encoded data of image -->
 </image>
</vmware>
```

The **FETCH** and **EMBED** types are mapped to a **BLOB** type through the client API (the mechanism of retrieving the value is hidden from the client which merely requests a **BLOB** type through the API). The **STREAM** type is visible at the API and is handled via special access methods.

2-5-3 Encoding

The encoding of a property element. This attribute cannot be used with resource elements. It defines the way the literal value is embedded. One encoding type might be Base64 for example. The permissible encoding types are specified in the DTD.

The encoding of a property element is defined using the **cfg:encoding** attribute. If omitted the encoding defaults to **LITERAL** (parsed character data) as specified in the DTD.

2-5-4 Issue

Timestamps mean elements can be optionally traversed based on whether they have changed since the last traversal.

Like directories and files in a file system, resource and property elements can be timestamped. This is achieved by assigning elements an issue number with the **cfg:issue** attribute.

```
<vmware cfg:issue="20010402130248">
 <encrypt cfg:issue="20010402130132">no</encrypt>
 <samba>yes</samba>
</vmware>
```

In a simple increasing numeric series "now" is always larger than any other value (i.e. an empty value would be an infinite value).

The value of the **cfg:issue** attribute is just a simple number. The format of the issue number is an increasing numeric sequence constructed from the date / time of the change, that is: **CCYYMMDDhhmmss**.

The issue number of a resource is expected to be increased whenever the issue number of any property or resource contained by that resource is increased. Issue numbers cannot decrease.

This attribute is not mandatory so the client node should interpret any element missing one as inheriting the timestamp of its parent and if there is no timestamp up the whole tree to the root node the timestamp is taken to be now (i.e. the elements timestamp would always be the current time), so in the above example the timestamp of the **samba** property would be the same as the timestamp of the **vmware** resource. The timestamp of a property cannot ever be more recent than that of its containing resource, as in the **encrypt** property above.

2-5-5 Derivation

The derivation of a resource or property element describes how the value of that element was obtained from the high level description. Essentially it is a list of classes. Primarily used for debugging purposes.

The derivation of any particular element can be recorded using the **cfg:derivation** attribute. The actual value of the attribute is open and not part of this specification.

2-5-6 Template

The template of a resource or property element is for custom extension. Initially it is used only to encode some additional mapping for LCFG ported resources (specifying how to reconstruct tagged entries).

The description of an element can be assigned using the **cfg:template** attribute. The actual value of the attribute is open and not part of this specification.

This is a temporary attribute required for the adaptors of the existing LCFG resources. It is unlikely to appear in the final specification.

2-5-7 Access Control

Access to resource and property elements is controlled by providing a list of users (or more generally authorization principles) allowed access to read the resource or property and a list of users (or more generally authorization principles) denied access to read the resource or property.

The **cfg:access** attribute is used to control which users can read the resource or property element it is applied to. The actual value of the attribute is open and the access control mechanism itself and any inheritance mechanism is undefined and is not part of this specification. The example below uses a simple inclusion / exclusion list to illustrate a possible syntax.

```
<vmware cfg:access="user1:include; user2:include;
user3:include; user4:include">
 <encrypt cfg:access="user2:exclude">no</encrypt>
 <samba cfg:access="user5:include; user1:exclude">
   yes
 </samba>
</vmware>
```

In the above example the default access for the **vmware** resource and any property in the resource allows **user1**, **user2**, **user3** and **user4** to read the elements. However **user2** is denied access to read the **encrypt** property and **user1** is denied access to read the **samba** property. Additionally **user5** can access the **samba** property (but no other property in the **vmware** resource or the resource itself).

2-5-8 Context Switching

Context switching (handling multiple states on the client machine such as connected and disconnected operation) is supported by using an attribute holding a literal list of context names for which the value applies under. Resources and property elements can then be specified with different values for different contexts. The context(s) of an element are defined using the **cfg:context** attribute.

This is the only circumstance where multiple instances of the same element can occur without a container (but they must each have a context attribute as specified in the DTD as a requirement).

```
<vmware>
 <encrypt cfg:context="connected">yes</encrypt>
 <encrypt cfg:context="disconnected">no</encrypt>
 <samba>yes</samba>
</vmware>
```

In the above example the value of the **encrypt** property element is dependent on the context (determined dynamically on the client node). A context can also be applied to a resource element.

```
<vmware cfg:context="disconnected">
    <encrypt>yes</encrypt>
    <samba>yes</encrypt>
</vmware>
<vmware cfg:context="connected">
    <encrypt>no</encrypt>
    <samba>no</encrypt>
</vmware>
```

Resource or property elements left out of a specific context apply in all contexts. In the above if the context state is not connected or disconnected the **encrypt** and **samba** properties (and indeed the **vmware** resource) would not exist in the profile. This cannot be validated by the DTD. If a resource or property element can exist in multiple contexts then the **cfg:context** attribute is a requirement (the element cannot be specified without it) and this is specified in the DTD.

Expanded values might often be used in some contexts to supply a value for a property which is dynamically determined on the client node (from the system or user environment).

```
<network cfg:context="home">
    <ipaddress>${ISPIP}</ipaddress>
</network>
<network cfg:context="work">
    <ipaddress>129.215.1.1</ipaddress>
</network>
```

The client node can only be in one context state at a time although that one context state can be made up of multiple individual context names. For example, if the contexts *connected/disconnected* and *home/work* existed the client node could be in one of four possible context states. An example of what the node profile for this might look like is shown below.

```
<network cfg:context="connected&home">
    <ipaddress>${ISPIP}</ipaddress>
</network>
<network cfg:context="connected&work">
    <ipaddress>129.215.1.1</ipaddress>
</network>
<network cfg:context="disconnected&(home|work)">
    <ipaddress>NULL</ipaddress>
</network>
```

The nature and syntax of the **cfg:context** attribute value and mechanism for specifying multiple contexts is open and not part of this specification. The example above shows one possible syntactic convention.

A Example DTD

Like the XML profile the DTD would also be generated automatically. However a very small example is included below. The DTD is for validation of the following resource tree.

- A **DISK** resource containing a **DEVICE** property and a **PARTITIONS** property represented as a nested resource.
- A **PARTITIONS** resource containing multiple instances of the **PARTITION** property represented as a nested resource.
- A **PARTITION** resource containing a **SIZE** property and a **MOUNT** property.

Below is the XML for an example node view profile for the above tree which includes a DTD which validates it.

```
<?xml version="1.0"?>
<!DOCTYPE profile [
 <!ELEMENT profile (disk)>
 <!ELEMENT disk (device,partitions)>
 <!ELEMENT device (#PCDATA)>
 <!ELEMENT partitions (partition*)>
 <!ELEMENT partition (size,mount)>
   <!ATTLIST partition cfg:name CDATA #REQUIRED>
 <!ELEMENT size (#PCDATA)>
 <!ELEMENT mount (#PCDATA)>
]>
<profile>
 <disk>
   <device>/dev/hda</device>
   <partitions>
     <partition cfg:name="root">
       <size>1000</size>
       <mount>/</mount>
     </partition>
     <partition cfg:name="tmp">
       <size>250</size>
       <mount>/tmp</mount>
     </partition>
   </partitions>
 </disk>
</profile>
```

Note that the DTD doesn't define all the attributes, only the single attribute used in the example. Note also that SGML DTD's do not support XML-Namespace so the namespace declarations have been stripped.

Full and proper validation is likely to require the use of XML-Schema.