Pp. 83-98 of the *Proceedings of LISA '02: Sixteenth Systems Administration Conference* (Berkeley, CA: USENIX Association, 2002).

Pan: A High-Level Configuration Language

Lionel Cons and Piotr Poznański – CERN, European Organization for Nuclear Research

ABSTRACT

The computational requirements for the new Large Hadron Collider are enormous: 5-8 PetaBytes of data generated annually with analysis requiring 10 more PetaBytes of disk storage and the equivalent of 200,000 of today's fastest PC processors. This will be a very large and complex computing system, with about two thirds of the computing capacity installed in "regional computing centres" across Europe, America, and Asia.

Implemented as a global computational grid, the goal of integrating the large geographically distributed computing fabrics presents challenges in many areas, including: distributed scientific applications; computational grid middleware, automated computer system management; high performance networking; object database management; security; global grid operations.

This paper describes our approach to one of these challenges: the configuration management of a large number of machines, be they nodes in large clusters or desktops in large organizations.

Introduction

The European Organization for Nuclear Research (CERN) is building the Large Hadron Collider (LHC), the world's most powerful particle accelerator. From the LHC Computing Grid Project home page (http://cern.ch/LHCgrid):

> *The computational requirements of the experiments that will use the LHC are enormous: 5-8 PetaBytes of data will be generated each year, the analysis of which will require some 10 PetaBytes of disk storage and the equivalent of 200,000 of today's fastest PC processors. Even allowing for the continuing increase in storage densities and processor performance this will be a very large and complex computing system, and about two thirds of the computing capacity will be installed in "regional computing centres" spread across Europe, America and Asia.*
>
> *The computing facility for LHC will thus be implemented as a global computational grid [10], with the goal of integrating large geographically distributed computing fabrics into a virtual computing environment. There are challenging problems to be tackled in many areas, including: distributed scientific applications; computational grid middleware, automated computer system management; high performance networking; object database management; security; global grid operations.*

This paper describes our approach to one of these challenges: the configuration management of a large number of machines, be they nodes in large clusters or desktops in large organizations.

Large Scale System Administration

Many solutions for managing a few machines do not scale well. When dealing with thousands of machines, some problems start to overwhelm.

Automation

It is fine to reinstall your home PC by booting from an installation diskette and typing a few commands but you certainly do not want to do it on a large cluster. Similarly, it is acceptable to log into one machine and purge /tmp by hand but this could also be automated using a tool like Red Hat's tmpwatch.

Technical solutions exist in many domains (remote power control, console concentration, network booting, unattended system installation, package management, etc.) so manual interventions can and should be limited to the absolute minimum. What remains to be done by hand is to configure the programs that will automate these otherwise manual tasks.

This approach has an interesting side effect. System configurations are known to rot with time because ad hoc system interventions tend to accumulate small mistakes until the system malfunctions. An unattended but complete reinstallation from scratch (not a restore from backup) is the most cost effective way to get rid of the problem. All you need is to make sure that the configuration of the installer is kept up to date, which is not difficult to do.

Abstraction

Once we have reached the full automation of the installation process, we can define the configuration of a machine as the sum of all the configurations of all the programs used either during the installation or afterwards. This will contain everything from disk partitioning information to system or software configuration (networking, user accounts, X server, etc.).

You should not include all the things that *can* be configured but rather the ones that *will* be configured. For instance, if you do not need to change the /usr/share/magic file used by the file command, consider it as a (static) data file that comes with the file package itself and therefore outside of the machine configuration abstraction.

We then want to reason about these machine configurations and, for instance, express the fact that two machines have the same disk model or the fact than one thousand machines belong to the same batch cluster. It is tedious to use the native configuration files, e.g., /etc/services to describe the known network services or /etc/crontab for cron's configuration, because of the duplication of information and the variety of convoluted formats. We really need an abstraction of this information that is easy to use.

The way the configuration information is abstracted and represented is very important. This is in line with Eric S. Raymond's advice in *The Cathedral and the Bazaar* [16]: "Smart data structures and dumb code works a lot better than the other way around."

It is quite expensive to come to a good abstraction but it really pays off when managing many machines. It is a virtuous circle: the more data you put in the abstraction, the more useful it becomes.

Single Database for Multiple Tools

In theory, a unique system administration tool is better than a set of unrelated and often overlapping tools such as AutoRPM, cfengine, RDist, LCFG, etc. In practice, such a mythical beast does not exist and system administrators use the tools' combination which is adapted to their needs, often with a pinch of home made scripts with "glue languages" such as Perl or TCL.

It is good to combine the strengths of these tools, but the variety of their configuration formats is a big disadvantage. Information is duplicated and often cumbersome to maintain. Until recently, the machines in our computer centre were drawing on information from more than twenty different sources, from flat files to real databases. Mistakes when handling these files (e.g., adding a machine and forgetting to update one file) were a common source of problems.

A good approach is to use a single source of information (a central configuration database) and simple programs that can transform this information into the format understood by the tools used.

Change Management

In the first eight months of its life, Red Hat Linux 7.2 had 311 updated RPMs. This is more than one updated RPM per day on average. Just looking at security, Red Hat issued 59 security advisories for the same system during the same period, almost two per week on average. In large computing centres, changes will occur frequently so you must manage them adequately.

Every component (be it hardware or software) has a non-zero failure probability so, statistically, large computer centres have a high probability of having, at any point in time, one or more components not working properly. This is especially true when using cheap commodity hardware. Some machines will always be down or somehow unreachable, so configuration changes have to be deployed asynchronously.

Moreover, some critical processes cannot be interrupted and intrusive system management tasks (such as changing the kernel in use) have to be deferred until the system is ready to accept the changes.

The consequence is that you should not try to configure a machine directly but rather change its configuration (stored outside of the machine) and let the machine bring itself into line whenever it can. This can be called convergent or asymptotic configuration (see for instance [18]): the machines independently try to come closer to their "desired state."

Validation

We should keep in mind this slight modification of one of Murphy's laws: anything that can go wrong will go wrong more spectacularly with central system administration.

Having a central database holding machine configurations and letting thousands of programs on remote machines use it is very powerful but mistakes can have disastrous consequences. It is of paramount importance to control the changes and detect mistakes before it is too late. Advanced means of validating the stored information must be in place.

The good news is that the abstraction mentioned above really helps. Once you can reason about machines and their configuration parameters, it is easy to express constraints such as "for all the machines, the filesystems mounted through NFS must be exported by the corresponding server." The example given in Appendix C describes exactly this constraint in our Pan language.

Validation should not only be seen as a way to prevent mistakes, it can also be used to make sure that things are really the way you want them to be. Constraints can be used, for example, to ensure that all machines have enough swap space, or that they are running the correct version of some software.

Our Solution

Overview

The Fabric Management Work Package (WP4 [21]) of the European Union DataGrid Project (EDG [8]) seeks to control large computing fabrics through the central management of their "desired state" via a central configuration database (one per administrative domain). This information will then be used in different ways.

For the initial system installation (we currently use Red Hat Linux 7.2), it will be used to create the various files needed to fully automate this process, for instance DHCP entries and Kickstart files.

For the system maintenance (we currently use LCFG), the configuration information will be directly used by a number of modular ''component'' scripts which are responsible for different subystems, such as ''mail configuration'' or ''web server configuration.'' The components are notified when their configuration changes and are responsible for translating the abstract configuration into the appropriate configuration files, and reconfiguring any associated daemons.

Machines will be self healing thanks to sensors reporting information to a monitoring database and actuators using this information to trigger recovery actions such as restarting a daemon or, in extreme cases, triggering a full reinstallation of the machine. Through the inclusion of hardware information in the configuration database, we can also detect problems such as a dead CPU or stolen memory.

Configuration Database

The configuration database [4] stores two forms of configuration information. One is called the High Level Description [5] and is expressed in the Pan language. The other is the Low Level Description [11] and is expressed in XML. Both are explained below.

The system administrators can edit the High Level Description, either directly or through some scripting layer. The Low Level Description (one XML file per machine) is always generated using the Pan compiler.

The XML machine configuration is cached on the machine (to support disconnected operations) and access is provided through a high-level library [15] that hides the details such as the XML schema used.

The database itself includes a scalable distribution mechanism for the XML files based on HTTP, and the possibility of adding any number of backends (such as LDAP or SQL) to support various query patterns on the information stored. It should scale to millions of configuration parameters.

Low Level Description

Mapping a configuration abstraction to a tree structure is quite easy. This is the natural format for most ''organized information,'' from files in a filesystem to the Windows registry or LDAP. We call this the Low Level Description (LLD).

Simple values (like strings or numbers) form the leaves of this tree and are called *properties*. Internal nodes of the tree are called *resources* and are used to group *elements*[1] into *lists* (accessed by index) or named lists (aka *nlists*, accessed by name). Nlists can conveniently be used to represent tables or records[2]. Every element has a unique *path* which identifies its position in the tree.

We chose XML to represent this tree in a file because it maps well to the hierarchical structure of the information and it is easy to parse and to validate (with XML Schema). Here is a small example representing some hardware information (a larger example can be found in Appendix A):

```
<?xml version="1.0" encoding="utf-8"?>
<nlist name="profile">
  <nlist name="hardware">
    <nlist name="memory">
      <long name="size">512</long>
    </nlist>
    <list name="cpus">
      <nlist>
        <string name="vendor">
          Intel
        </string>
        <string name="model">
          Pentium III (Coppermine)
        </string>
        <double name="speed">
          853.22</double>
      </nlist>
    </list>
  </nlist>
</nlist>
```

Starting with the toplevel XML element (named ''profile''), you can see on the fifth line the property describing the memory size: its path is /hardware/memory/size and its value is the long integer 512. Similarly, /hardware/cpus is the resource representing the list of CPUs. The path /hardware/cpus/0[3] identifies the first (and only) CPU which is represented using a nlist that holds a kind of record or structure describing the CPU. The model of the CPU is the string at path /hardware/cpus/0/model and its value is ''Pentium III (Coppermine).''

The way this information appears in the XML file is dependent on the programs using it. For instance, if you have only a few X server configurations but a large variety of resolution settings, you could have something like:

```
<nlist name="profile">
  <nlist name="system">
    <nlist name="x">
      <string name="XF86Config"
          type="fetch">
http://config.cern.ch/XF86Config-ATI64-19
      </string>
      <list name="modes">
        <string>1280x1024</string>
        <string>1024x768</string>
      </list>
    </nlist>
  </nlist>
</nlist>
```

The special *fetch* type is known by our system and the programs accessing the configuration information through our API will simply see the contents of the file at the given URL as a string. A program

[1] The term *element* refers to either a property or a resource.
[2] I.e., similar to Pascal's record or C's struct.
[3] In paths, numbers are used to identify list items, the first one having the index 0, like in C.

responsible for managing the X configuration would simply have to start with this base file, substitute in the desired modes and write the result to /etc/X11/XF86Config.

High Level Description

Although the previous XML representation is sufficient for the programs running on the target machines, we need a High Level Description (HLD) to reason about groups of machines and share common information.

Existing tools (such as m4) only cover some of our requirements so we decided to design our own language to represent the HLD and we wrote the accompanying compiler transforming this HLD into LLD (i.e., XML).

We believe (in line with Paul Anderson's *A Declarative Approach to the Specification of Large-Scale System Configurations* [2]) that a declarative approach[4] to configuration specification is better suited than a procedural one[5]. Pan has been designed to stay as declarative as possible while allowing some form of procedural code, which is required to take full advantage of the power of validation.

The following is a quick overview of the salient features of the Pan language. The complete language specification is available in another document [5].

[4] I.e., describe how things should look like in the end.
[5] I.e., describe the sequence of actions to be performed.

Overview

Pan mainly consists of *assignments*, each of which sets some value in a given part of the LLD identified by its path. The following code can be used to generate the LLD shown earlier. The left hand side of the assignment is the path and the right hand side is the value. nlist is a builtin function that will return the nlist made from its arguments.

```
"/hardware/memory/size" = 512;
"/hardware/cpus/0" = nlist(
  "vendor", "Intel",
  "model",  "Pentium III (Coppermine)",
  "speed",   853.220,
);
```

Pan also features other statements like include (very similar to cpp's #include directive) or delete that can delete a part of the LLD.

The grouping of statements into *templates* allows the sharing of common information and provides a simple inheritance mechanism. A *structure template* is used to represent a subtree of information (for instance a given disk) while an *object template* represents a real world object (the compiler will generate a separate LLD for every object template encountered).

Listing 1 shows a partial example of two cluster nodes that share most of their configuration information.

Types

Pan contains a very flexible typing mechanism. It has several builtin types (such as boolean, string, long,

```
# definition for the disk IBM DTLA-307030
structure template disk_ibm_dtla_307030;
"type"     = "disk";
"vendor"   = "IBM";
"model"    = "DTLA-307030";
"size"     = 29314; # MB

# definition for the hardware Elonex 800x2/512
structure template pc_elonex_800x2_512;
"vendor"       = "Elonex";
"model"        = "800x2/512";
"cpus"         = list(create("cpu_intel_p3_800"), create("cpu_intel_p3_800"));
"memory/size"  = 512; # MB
"devices/hda"  = create("disk_ibm_dtla_307030");

# definition for the Venus cluster
template cluster_venus;
"/hardware" = create("pc_elonex_800x2_512");
# and any other hardware or system information shared by all the
# members of the Venus cluster

# first machine
object template venus001;
include cluster_venus;
"/hardware/serial" = "CH01112041";

# second machine
object template venus002;
include cluster_venus;
"/hardware/serial" = "CH01117031";
# the first disk has been replaced
"/hardware/devices/hda" = create("disk_quantum_fireballp_as20_5");
```

Listing 1: Two cluster nodes which share most of their configuration.

. . .) and allows compound types to be built on top of these. Once the type of a configuration element is known, the compiler makes sure that only values of the right type are assigned to it. By explicitly specifying the type of the root element (i.e., the top of the configuration tree), one can completely define the schema of the information that is found in the LLD. The type enforcement done by the compiler guarantees that only LLDs conforming to the schema will be generated. This enforcement is illustrated in Listing 2.

Starting with the root of the LLD, the compiler will make sure that the data corresponds to the declared type. Extra and missing fields in structures trigger a compilation error. The code in Listing 2 ensures that /hardware/memory/size is always present and contains a positive long integer.

Validation

To have even greater control on the information generated by the compiler, one can attach arbitrary validation code either to a type or to a configuration path; see Listing 3.

Data Manipulation Language

The validation code is represented in a simple yet powerful data manipulation language which is a subset of Pan and syntactically similar to C or Perl. Rather than embedding another language such as Perl or Python for this task, we decided to design our own. This was necessary to maintain control over type checking and to encourage users to use the declarative parts of Pan. Builtin functions such as pattern matching and substitution are available and user defined functions are supported.

Although we prefer the declarative approach to the procedural approach, this data manipulation language is very convenient to perform complex operations. Listing 4 illustrates the use of Pan to introduce an element into a given list position.

```
# structure representing the (physical) memory
define type memory_t = {
  "size" : long(0..)       # a long which is greater than 0
};

# structure representing the complete hardware
define type hardware_t = {
  "vendor"  : string
  "model"   : string
  "serial"  ? string       # this field is optional
  "memory"  : memory_t
  "cpus"    : cpu_t[1..8]  # a list of between 1 and 8 cpu_t
  "devices" : device_t{}   # a (maybe empty) table of device_t
};

# structure representing the root of the configuration tree
define type root_t = {
  "hardware" : hardware_t
  "system"   : system_t
  "software" : software_t
};

# the root of the configuration tree (i.e., /) must be of type root_t
type "/" = root_t;
```

Listing 2: Type enforcement.

```
# IPv4 address in dotted number notation
define type ipv4 = string with {
  result = matches(self, '^(\d+)\.(\d+)\.(\d+)\.(\d+)$');
  if (length(result) == 0)
    return("bad string");
  i = 1;
  while (i <= 4) {
    x = to_long(result[i]);
    if (x > 255)
      return("chunk " + to_string(i) + " too big: " + result[i]);
    i = i + 1;
  };
  return(true);
};

# make sure that we have at least 256MB of RAM per processor
valid "/hardware/memory/size" = self >= 256 * length(value("/hardware/cpus"));
```

Listing 3: Validation code.

Miscellaneous

The Pan compiler keeps track of derivation information which precisely links configuration information appearing in the LLD to the originating HLD statements. When HLD templates are modified, this derivation information is used to determine which LLDs must be recreated, thus minimizing the work carried out by the compiler. The information is also used to determine which HLD templates are responsible for the final value of a given configuration parameter.

Comparison With Other Tools

Pan (and its associated compiler) cannot be considered as a system administration tool by itself: it is only a language to express configuration information. As far as we know, there exists no similar tool to compare directly with. What follows is a comparison with the way different tools or projects manipulate system configuration information.

Arusha Project

The Arusha Project (ARK, http://ark.sourceforge.net) provides a framework for collaborative system administration [13]. It provides a simple, XML-based language that can be used to describe almost everything, from package management to documentation or system configuration. Unfortunately, this language lacks strong type checking and validation. It also mixes code and data (e.g., some Perl or Python code can be embedded inside XML, close to the data) which is something that we do not want.

Cfengine

Cfengine (http://www.cfengine.org) is an autonomous agent [3] with a high level declarative language to manage large computer networks. It has no real types and a limited support for lists. Its configuration is not a real abstraction of the machine configuration but rather some instructions for its different modules such as network interface configuration, symbolic links management, checks for permissions and ownership of files, etc. For instance, the modification of system files like /etc/inetd.conf is often done with instructions such as AppendIfNoSuchLine or CommentLinesMatching. It has no support for validation.

DMTF

The Distributed Management Task Force (DMTF, http://www.dmtf.org) is an organization developing "management standards." The standards closest to Pan are part of the Common Information Model (CIM, http://www.dmtf.org/standards/standard_cim.php). Their approach is complex and mixes configuration management and system monitoring. Although their standards could not be used directly inside our work, we have tried to stay close and to reuse some parts of their data schemas.

LCFG

LCFG [1] (http://www.lcfg.org) is a system for automatically installing and managing the configuration of large numbers of Unix systems. It does use some abstraction to describe the machine configuration but the language used does not really have types. All the parameters are basically strings similar to X resources and compound types (such as lists or tables) are built on top of these with some ad-hoc name mangling. The inheritance is achieved by using cpp and include files. The combination of cpp macros and embedded Perl code hinder the clarity of this otherwise mainly declarative language. On the other hand, it has some advanced features (like constraint based list ordering) that will probably be added to Pan in the future.

Although LCFG and EDG are separate projects, the development teams share ideas and some compatibility exists. For instance, the Pan compiler can produce some XML files that can be understood by the LCFG components.

```
# insert a string after another one in a list of strings
# (or at the end if not found)
define function insert_after = {
  if (argc != 3 || !is_string(argv[0]) || !is_string(argv[1]) ||
      !is_list(argv[2]))
    error("usage: insert_after(string, string, list)");
  idx = index(argv[1], argv[2]);
  if (idx < 0) {
    # not found, we insert at the end
    splice(argv[2], length(argv[2]), 0, list(argv[0]));
  } else {
    # found, we insert just after
    splice(argv[2], idx+1, 0, list(argv[0]));
  };
  return(argv[2]);
};

# here is how to use it to insert "apache" after "dns"
"/boot/services" = list("dns", "dhcp", "mail", "postgres");
"/boot/services" = insert_after("apache", "dns", value("/boot/services"));
```

Listing 4: Introducing an element into a given list position.

Status and Availability

After a first Perl prototype last year, the new compiler (built using C++, STL, Lex and Yacc) is almost complete (at the time of this writing) and will be delivered to the EDG in September.

The software will be available under the open source EDG Software License[6] from the EDG WP4 Configuration Task web site at http://cern.ch/hep-proj-grid-fabric-config .

At the time of this writing, the Pan language has been successfully used to describe a large fraction of the configuration of the Linux machines used inside the EDG project. Work is in progress to extend this to other machines in our computer centre at CERN.

Acknowledgements

We would like to thank the European Union for their support of the EDG project and our colleagues from the WP4 for their contributions and very fruitful discussions on the topics of system administration and configuration.

Authors Biographies

Lionel Cons earned an "Ingénieur de l'École Polytechnique" (Paris) diploma in 1988 and the ENSIMAG's engineer's diploma two years later. He then joined CERN where he worked as a C software developer and then as a UNIX system engineer in the Information Technology division. He presently works on system security and is the leader of the WP4 configuration task of the EDG project.

Piotr Poznański earned an MSc degree in Computer Science from the Univeristy of Mining and Metallurgy (Cracow, Poland). He joined CERN in 2000 and currently works in the EDG project as a software engineer.

References

[1] Anderson, Paul, "Towards a High-Level Machine Configuration System," *LISA Conference Proceedings*, 1994.

[2] Anderson, Paul, *A Declarative Approach to the Specification of Large-Scale System Configurations*, http://www.dcs.ed.ac.uk/home/paul/publications/conflang.pdf, 2001.

[3] Burgess, Mark,"Computer Immunology," *LISA Conference Proceedings*, 1998.

[4] Cons, Lionel and Piotr Poznański, *Configuration Database Global Design*, http://cern.ch/hep-proj-grid-fabric-config, 2002.

[5] Cons, Lionel and Piotr Poznański, *High Level Configuration Description Language Specification*, http://cern.ch/hep-proj-grid-fabric-config, 2002.

[6] da Silva, Fabio Q. B., Juliana Silva da Cunha, Danielle M. Franklin, Luciana S. Varejao, and Rosalie Belian, "A Configuration Distribution System for Heterogeneous Networks," *LISA Conference Proceedings*, 1998.

[7] da Silveira, Gledson Elias and Fabio Q. B. da Silva, "A Configuration Distribution System for Heterogeneous Networks," *LISA Conference Proceedings*, 1998.

[8] *European Union DataGrid Project (EDG)*, http://www.eu-datagrid.org .

[9] Evard, Rémy, "An Analysis of UNIX System Configuration," *LISA Conference Proceedings*, 1997.

[10] Foster, Ian and Carl Kesselman, *The Grid: Blueprint for a New Computing Infrastructure*, Morgan Kaufmann, http://www.mkp.com/books_catalog/catalog.asp?ISBN=1-55860-475-8, 1998.

[11] George, Michael, *Node Profile Specification*, http://cern.ch/hep-proj-grid-fabric-config, 2002.

[12] Harlander, Dr. Magnus, "Central System Administration in a Heterogeneous Unix Environment: GeNUAdmin," *LISA Conference Proceedings*, 1994.

[13] Holgate, Matt and Will Partain, "The Arusha Project: A Framework for Collaborative Unix System Administration," *LISA Conference Proceedings*, 2001.

[14] *Large Scale System Configuration Workshop*, http://www.dcs.ed.ac.uk/home/dcspaul/wshop, 2001.

[15] Poznański, Piotr, *Node View Access API Specification*, http://cern.ch/hep-proj-grid-fabric-config, 2002.

[16] Raymond, Eric S., *The Cathedral and the Bazaar*. O'Reilly, http://www.oreilly.com/catalog/cb, 1999.

[17] Rouillard, John P. and Richard B. Martin, "Config: A Mechanism for Installing and Tracking System Configurations," *LISA Conference Proceedings*, 1994.

[18] Sventek, Joe, *Configuration, Monitoring and Management of Huge-scale Applications with a Varying Number of Application Components*, http://www.dcs.ed.ac.uk/home/dcspaul/wshop/HugeScale.pdf, 2001.

[19] Traugott, Steve and Joel Huddleston, "Bootstrapping an Infrastructure," *LISA Conference Proceedings*, 1998.

[20] van der Hoek, André, Dennis Heimbigner, and Alexander L. Wolf, *Software Architecture, Configuration Management, and Configurable Distributed Systems: A Ménage à Trois*, http://citeseer.nj.nec.com/hoek98software.html, 1998.

[21] WP4, *EDG Fabric Management Work Package*, http://cern.ch/hep-proj-grid-fabric.

[22] WP4C, *EDG WP4 Configuration Task*, http://cern.ch/hep-proj-grid-fabric-config.

[6] http://www.eu-datagrid.org/license.html .

Appendix A: Partial LLD Example

This is an oversimplified example; more complete examples can be found on our web site [22].


```xml
<?xml version="1.0" encoding="utf-8"?>
<nlist name="profile" type="record">
  <nlist name="hardware" type="record">
    <string name="vendor">Elonex</string>
    <string name="model">850/256</string>
    <list name="cpus">
      <nlist type="record">
        <string name="vendor">Intel</string>
        <string name="model">Pentium III (Coppermine)</string>
        <double name="speed">853.22</double>
      </nlist>
    </list>
    <string name="serial">CH01112041</string>
    <nlist name="memory" type="record">
      <long name="size">256</long>
    </nlist>
    <nlist name="devices" type="table">
      <nlist name="hda" type="record">
        <string name="vendor">QUANTUM</string>
        <string name="model">FIREBALLP AS20.5</string>
        <string name="type">disk</string>
        <long name="size">19596</long>
      </nlist>
      <nlist name="hdc" type="record">
        <string name="vendor">LG</string>
        <string name="model">CRD-8521B</string>
        <string name="type">cd</string>
      </nlist>
      <nlist name="eth0" type="record">
        <string name="vendor">3Com</string>
        <string name="model">3c905B-Combo [Deluxe Etherlink XL 10/100]</string>
        <string name="type">net</string>
        <string name="driver">3c59x</string>
        <string name="address">00:d0:b7:a9:a3:47</string>
      </nlist>
    </nlist>
  </nlist>
  <nlist name="system" type="record">
    <list name="mounts">
      <nlist type="record">
        <string name="type">swap</string>
        <string name="path">swap</string>
        <string name="device">hda1</string>
      </nlist>
      <nlist type="record">
        <string name="type">ext2</string>
        <string name="path">/</string>
        <string name="device">hda2</string>
      </nlist>
      <nlist type="record">
        <string name="type">ext2</string>
        <string name="path">/var</string>
        <string name="device">hda3</string>
      </nlist>
      <nlist type="record">
        <string name="type">proc</string>
        <string name="path">/proc</string>
      </nlist>
      <nlist type="record">
        <string name="type">devpts</string>
        <string name="path">/dev/pts</string>
        <list name="options">
```

```
          <string>gid=5</string>
          <string>mode=620</string>
        </list>
      </nlist>
      <nlist type="record">
        <string name="type">ext2</string>
        <string name="path">/mnt/floppy</string>
        <list name="options">
          <string>noauto</string>
          <string>owner</string>
        </list>
        <string name="device">fd0</string>
      </nlist>
      <nlist type="record">
        <string name="type">afs</string>
        <string name="path">/afs</string>
      </nlist>
      <nlist type="record">
        <string name="type">iso9660</string>
        <string name="path">/mnt/cdrom</string>
        <list name="options">
          <string>noauto</string>
          <string>owner</string>
          <string>ro</string>
        </list>
        <string name="device">hdc</string>
      </nlist>
    </list>
    <nlist name="partitions" type="table">
      <nlist name="hda1" type="record">
        <string name="type">primary</string>
        <string name="disk">hda</string>
        <long name="size">512</long>
        <long name="id">82</long>
      </nlist>
      <nlist name="hda2" type="record">
        <string name="type">primary</string>
        <string name="disk">hda</string>
        <long name="size">18828</long>
        <long name="id">83</long>
      </nlist>
      <nlist name="hda3" type="record">
        <string name="type">primary</string>
        <string name="disk">hda</string>
        <long name="size">256</long>
        <long name="id">83</long>
      </nlist>
    </nlist>
  </nlist>
</nlist>
```


Appendix B: Partial HLD Examples

These HLD templates have been used to generate the LLD found in Appendix A. More sample code can be found on our web site [22].

functions.tpl

```
################################################################################
#                Useful functions.
################################################################################
```

```
declaration template functions;

# insert_after(string, string, list): insert the first string after the second
# one (if found) or at the end (otherwise); the last argument is modified but
# also returned as the result of the function

define function insert_after = {
  if (argc != 3 ||
      !is_string(argv[0]) || !is_string(argv[1]) || !is_list(argv[2]))
    error("usage: insert_after(string, string, list)");
  idx = index(argv[1], argv[2]);
  if (idx < 0) {
    # not found, we insert at the end
    splice(argv[2], length(argv[2]), 0, list(argv[0]));
  } else {
    # found, we insert just after
    splice(argv[2], idx+1, 0, list(argv[0]));
  };
  return(argv[2]);
};

# given a disk name, return a table of three primary partitions for swap, root
# and /var with a very simple space allocation algorithm

define function simple_partitions = {
  if (argc != 1 || !is_string(argv[0]))
    error("usage: simple_partitions(string)");
  disk = argv[0];
  disk_size = value("/hardware/devices/" + disk + "/size");
  # swap is twice the size of the physical memory
  swap = nlist(
    "disk", disk,
    "type", "primary",
    "size", 2 * value("/hardware/memory/size"),
    "id",   82, # Linux swap
  );
  # var is 256MB for disks larger than 2GB, 128MB otherwise
  var = nlist(
    "disk", disk,
    "type", "primary",
    "size", if (disk_size > 2048) 256 else 128,
    "id",   83, # Linux
  );
  # root is the rest
  root = nlist(
    "disk", disk,
    "type", "primary",
    "size", disk_size - swap["size"] - var["size"],
    "id",   83, # Linux
  );
  # order of partitions is swap, root and var
  return(nlist(
    disk+"1", swap,
    disk+"2", root,
    disk+"3", var,
  ));
};
```

types.tpl

```
################################################################################
# Useful (but simplified) types.
################################################################################
```

```
declaration template types;
##############################################################################
# simple types
# unsigned long
#(old style) define type ulong = long with self >= 0;
define type ulong = long(0..);
# unsigned double
define type udouble = double(0..);
# IPv4 address in dotted number notation
define type ipv4 = string with {
  result = matches(self, '^(\d+)\.(\d+)\.(\d+)\.(\d+)$');
  if (length(result) == 0)
    return("bad string");
  i = 1;
  while (i <= 4) {
    x = to_long(result[i]);
    if (x > 255)
      return("chunk " + to_string(i) + " too big: " + result[i]);
    i = i + 1;
  };
  return(true);
};
##############################################################################
# hardware types
# memory record
define type memory_t = {
  "size" : ulong
};
# CPU record
define type cpu_t = {
  "vendor" : string
  "model"  : string
  "speed"  : udouble
};
# device record (describing some hardware devices such as disks)
define type device_t = {
  "type"     : string with match(self, '^(disk|cd|net)$')
  "vendor"   : string
  "model"    : string
  "size"     ? ulong
  "driver"   ? string
  "address"  ? string
};
# hardware record (describing some complete hardware information)
define type hardware_t = {
  "vendor"  : string
  "model"   : string
  "serial"  : string
  "memory"  : memory_t
  "cpus"    : cpu_t[1..]   # list of at least one CPU
  "devices" : device_t{}   # table of devices, indexed by names such as hda
};
##############################################################################
# system types
# mount record (describing what will end up in /etc/fstab)
```

```
define type mount_t = {
  "device"  ? string
  "path"    : string
  "type"    : string
  "name"    ? string
  "options" ? string[]
};
# partition record (describing how to partition the disks)
define type partition_t = {
  "disk" : string with value("/hardware/devices/"+self+"/type") == "disk"
  "type" : string
  "size" : ulong
  "id"   : ulong
};
# system record (describing some of the system configuration)
define type system_t = {
  "mounts"     : mount_t[1..]   # list of at least one mount
  "partitions" ? partition_t{}  # table of partitions, indexed by, e.g., hda1
};
##############################################################################
# root type
# type of the root of the configuration information
define type root_t = {
  "hardware" : hardware_t    # hardware subtree
  "system"   : system_t      # system subtree
};
# declare that root is indeed of the root type
type "/" = root_t;
```

hardware.tpl

```
##############################################################################
# Sample hardware data.
##############################################################################
##############################################################################
# cpus
structure template cpu_intel_p3_800;
"vendor" = "Intel";
"model"  = "Pentium III (Coppermine)";
"speed"  = 796.550; # MHz
structure template cpu_intel_p3_850;
"vendor" = "Intel";
"model"  = "Pentium III (Coppermine)";
"speed"  = 853.220; # MHz
##############################################################################
# disks
structure template disk_quantum_fireballp_as20_5;
"type"     = "disk";
"vendor"   = "QUANTUM";
"model"    = "FIREBALLP AS20.5";
"size"     = 19596; # MB
structure template disk_ibm_dtla_307030;
"type"     = "disk";
"vendor"   = "IBM";
"model"    = "DTLA-307030";
"size"     = 29314; # MB
```

```
##############################################################################
# cdroms

structure template cdrom_lg_crd_8521b;
"type"     = "cd";
"vendor"   = "LG";
"model"    = "CRD-8521B";

##############################################################################
# network cards

structure template network_3com_3c905b;
"type"     = "net";
"vendor"   = "3Com";
"model"    = "3c905B-Combo [Deluxe Etherlink XL 10/100]";
"driver"   = "3c59x";

structure template network_intel_82557;
"type"     = "net";
"vendor"   = "Intel";
"model"    = "82557 [Ethernet Pro 100]";
"driver"   = "eepro100";

##############################################################################
# computers

structure template pc_elonex_850_256;
"vendor"       = "Elonex";
"model"        = "850/256";
"cpus"         = list(create("cpu_intel_p3_850"));
"memory/size"  = 256; # MB
"devices/hda"  = create("disk_quantum_fireballp_as20_5");
"devices/hdc"  = create("cdrom_lg_crd_8521b");
"devices/eth0" = create("network_3com_3c905b");

structure template pc_elonex_800x2_512;
"vendor"       = "Elonex";
"model"        = "800x2/512";
"cpus"         = list(create("cpu_intel_p3_800"), create("cpu_intel_p3_800"));
"memory/size"  = 512; # MB
"devices/hda"  = create("disk_ibm_dtla_307030");
"devices/eth0" = create("network_intel_82557");
```

system.tpl

```
##############################################################################
# Sample system data.
##############################################################################

##############################################################################
# standard mounts

structure template mount_afs;
"path"    = "/afs";
"type"    = "afs";

structure template mount_proc;
"path"    = "/proc";
"type"    = "proc";

structure template mount_devpts;
"path"    = "/dev/pts";
"type"    = "devpts";
"options" = list("gid=5", "mode=620");
```

```
structure template mount_floppy;
"device"  = "fd0";
"path"    = "/mnt/floppy";
"type"    = "ext2";
"options" = list("noauto", "owner");

structure template mount_cdrom;
"device"  = undef;
"path"    = "/mnt/cdrom";
"type"    = "iso9660";
"options" = list("noauto", "owner", "ro");

##############################################################################

# mounting templates

# add the standard Linux mount entries
template mounting_linux;
"/system/mounts" = merge(value("/system/mounts"), list(
  create("mount_proc"),
  create("mount_devpts"),
  create("mount_floppy"),
));

# add the AFS mount entry
template mounting_afs;
"/system/mounts" = merge(value("/system/mounts"), list(create("mount_afs")));
```

sample.tpl

```
##############################################################################

# Sample object template.

##############################################################################

object template sample;

# standard includes
include types;
include functions;

# hardware information
"/hardware" = create("pc_elonex_850_256");
"/hardware/serial" = "CH01112041";
"/hardware/devices/eth0/address" = "00:d0:b7:a9:a3:47";

# system information
"/system/partitions" = simple_partitions("hda");
"/system/mounts/0" = nlist("type", "swap", "path", "swap", "device", "hda1");
"/system/mounts/1" = nlist("type", "ext2", "path", "/",    "device", "hda2");
"/system/mounts/2" = nlist("type", "ext2", "path", "/var", "device", "hda3");
include mounting_linux;
include mounting_afs;

# we also add a mount entry for our CD drive ...
"/system/mounts" = merge(value("/system/mounts"),
                         list(create("mount_cdrom", "device", "hdc")));
# ...and make sure that hdc indeed contains a CD drive!
valid "/hardware/devices/hdc" = self["type"] == "cd";
```

Appendix C: NFS Validation Example

xvalidation.tpl

```
##############################################################################

# Simplified example of cross object validation.

# All the NFS clients check that the NFS servers that they use indeed export
# the directories that they mount. This is done transparently by adding some
```

```
# validation code to the mount record type. Further checks such as wildcards
# in export list or export/mount options mismatch are left as an exercise for
# the reader ;-)
# Here is how to compile the server and two clients (result on stdout):
#   % pan --stdout --output=nfssrv1 xvalidation.tpl            (will succeed)
#   % pan --stdout --output=nfsclt1 xvalidation.tpl            (will succeed)
#   % pan --stdout --output=nfsclt2 xvalidation.tpl            (will fail)
###############################################################################
###############################################################################
# types definitions
template types;
# export record (roughly what is in /etc/exports)
define type export = {
  "path"    : string   description "path of the exported directory"
  "client"  : string   description "name of client allowed to mount it"
  "options" ? string[] description "list of exporting options like ro"
};
# mount record (roughly what is in /etc/fstab)
define type mount = {
  "device"  : string   description "device as understood by the mount command"
  "path"    : string   description "path of the mount point"
  "type"    : string   description "type of the mounted filesystem"
  "name"    ? string   description "name or label of this mount entry"
  "options" ? string[] description "list of mounting options like ro"
} with valid_mount(self);
# validation of a mount record (only nfs type records are checked)
define function valid_mount = {
  # the mount record is our only argument
  mount = argv[0];
  # we only care about NFS mounts, other types are considered OK
  if (mount["type"] != "nfs")
    return(true);
  # the device field will give us the NFS server and path
  result = matches(mount["device"], '^([\w\.\-]+):(.+)$');
  if (length(result) == 0)
    error("bad nfs device: " + mount["device"]);
  server = result[1];
  path   = result[2];
  # we now look at the server's exports list
  exports = value("//" + server + "/system/exports");
  i = 0;
  len = length(exports);
  while (i < len) {
    # we check if this export record is good for us by checking the client
    # field against object (i.e., the name of the current object template)
    # and the path; we want exact match and ignore the export/mount options
    if (exports[i]["client"] == object && exports[i]["path"] == path)
      return(true);
    i = i + 1;
  };
  # we haven't found any export record matching our needs, we complain:
  error("server " + server + " does not export " + path + " to " + object);
};
###############################################################################
# NFS server definition
```

```
object template nfssrv1;

# type settings
include types;
type "/system/exports" = export[];

# data for this host
"/system/exports" = list(
  nlist(                    # we export /home to hostx
    "path",    "/home",
    "client",  "hostx",
  ),
  nlist(                    # we export /home to nfsclt1, read-only
    "path",    "/home",
    "client",  "nfsclt1",
    "options", list("ro"),
  ),
);

###############################################################################

# NFS clients definitions

template client;

# type settings
include types;
type "/system/mounts" = mount[];

# data for this host
"/system/mounts" = list(
  nlist(                    # we mount /dev/hda1 as the root filesystem
    "device", "/dev/hda1",
    "path",   "/",
    "type",   "ext2",
  ),
  nlist(                    # we NFS mount /home from the server nfssrv1
    "device", "nfssrv1:/home",
    "path",   "/home",
    "type",   "nfs",
  ),
);

# first client: known by the server, compilation will succeed
object template nfsclt1;
include client;

# second client: unknown to the server, compilation will fail with:
# *** user error: server nfssrv1 does not export /home to nfsclt2
object template nfsclt2;
include client;
```