Computer Sciences Department

A New Model for Managing Configuration Data

Adwait N. Tumbde
Michael M. Swift

Technical Report #1619

October 2007

A New Model for Managing Configuration Data

Adwait N. Tumbde and Michael M. Swift
University of Wisconsin-Madison
{adwait,swift}@cs.wisc.edu

Abstract

Configuration management is one of the largest causes of system and application failure. In one study, twenty four percent of Windows NT downtime was attributed to system configuration and maintenance [24]. Furthermore, system configuration is a large expense: 60-80% of the total cost of computer ownership is system management [6]. The problem is increasing as systems and applications get larger.

We seek to address a key aspect of this problem, *configuration storage*: how configuration data is stored and managed by the OS. Existing mechanisms, such as files in Linux, property lists in MacOS X, and the registry in Windows do not adequately support application and administrator needs. For example, common features such profiles must be implemented separately by each application. Settings for a single application are often stored in multiple locations, making it difficult to identify all the configuration state related to a single application.

To address these problems, we propose a new data model and storage service for configuration data. The Configuration Data Management System (CDMS) stores settings as name–value pairs, similar to other systems. However, CDMS can cluster related settings into *objects* that allow the settings for an application to be centralized, even if the settings names are widely distributed. CDMS assembles *configuration spaces* from a list of objects to support profiles generically and to support simultaneous use of multiple versions of an application. We present an implementation of CDMS and present case studies of using CDMS to manage configuration data for Mozilla Firefox and Apache.

1 Introduction

Modern operating systems are incredibly flexible, but this flexibility comes at a large cost: configuration management. Configuration management adversely impacts both system availability and the cost of ownership. The Computing Research Association reports that 60-80% of the cost of ownership is due to system management [6]. A study of Windows NT systems indicated that configuration management was responsible for 24% of downtime [24]. Common configuration problems arise when an install or uninstall process fails, when an application or administrator corrupts data, or when an upgrade overwrites common configuration settings incorrectly [12].

A major contributor to the difficulty of system management is the *organization* of configuration data. For example, Windows XP stores settings in a configuration registry, a hierarchical database of key-value pairs. A typical machine has approximately 200,000 settings [14]. However, additional information of use to administrators, such as default values, schema, and comments are not available in this model. Furthermore, the rigid hierarchical structure prevents applications that interact from associating their joint configuration with both applications. Conversely, Unix systems traditionally store data in application-defined text file formats. This provides flexibility to applications but complicates management, as each application requires a separate parser.

Furthermore, application developers and administrators construct higher-level services on top of configuration storage, but must do so in an ad-hoc, application-specific fashion. For example, applications commonly implement an inheritance model, where users may have per-user settings that override system-wide defaults. As another example, many applications implement *profiles*, a grouping of settings that may be selected en masse. Finally, applications interested in robustness must implement transactions and rollback when modifying configuration settings, to ensure that they can recover after a failure. Unfortunately, such improvised services raise the complexity of configuration management. These services remove the semantics of configuration from the data, because applications now use their private logic to construct internal settings from the stored data.

We seek to simplify the job of programmers and administrators by offering a better configuration storage mechanism. This mechanism simplifies many of the services currently provided by applications, supports common administration tasks, and ensures that the configuration state of an application is visible to administrators and not hidden behind a layer of application semantics.

Our Approach. In this paper we present a new data model for configuration settings that simplifies many management tasks. The model support wide variety of configuration uses, such as OS settings, application settings, and user preferences. Our data model centers around three abstractions. *Properties* are name–value pairs in a global namespace. The names have a hierarchical structure. The data model also groups related settings into *configuration objects*. This allows grouping of settings related by use (for example, relevant to a single application) even if those settings would normally

stored in different parts of a configuration hierarchy. Further, objects may be assembled into *configuration spaces*, which are collections of settings visible to an application instance. Spaces provide inheritance, in which objects are ordered into a list and attributes specified at the front of the list override attributes further down the list. This provides generic support for profiles, by constructing a different space for each profile, as well as simultaneous support for multiple versions of an application (again, with a space for each version).

We implemented the data model in the Configuration Data Management System (CDMS), which relies on a relational database to store and query configuration data. While a relational database is not vital to the data model, its existing support for transactions and queries greatly simplified development. This service supports operations that applications must currently implement, such as transactions, inheritance, and profiles. Additionally, CDMS provides a rich query interface and a log of all persistent changes.

To gain experience with CDMS, we use it to store the configuration settings for two application, the Mozilla Firefox web browser and the Apache web server. We find that CDMS easily stores Firefox data and provides profile support without special application logic. It also supports a rich management model, in which administrators may decide exactly which settings may be overridden by users. We also show how Apache's less-structured data also fits well within CDMS.

The next section describes the shortcomings of existing configuration storage systems. We present related work in Section 3. Section 4 details the our data model, and Section 5 describes the architecture and implementation of CDMS. We present case studies in Section 6. We finish with a comparison to using files for configuration data in Section 7 and concluding remarks in Section 8.

2 The Problem

We find ample evidence in modern operating systems that existing methods of storing configuration data are inadequate. While we primarily discuss Windows and Linux, we find problems in MacOS X as well.

Ad-hoc representation. In Linux, there is no single configuration service, leading to hundreds of application-specific data formats. This variety leads to formatting errors when changing settings. While Windows supports a common storage system, the registry, it does not provide high level features such as naming. Applications that link settings to other applications use their own naming convention, such as globally unique identifiers (GUIDs) to indirectly reference other applications. However, the context for looking up these strings is known only to the application, which conceals the semantics of these links from administrators. This usage creates implicit dependencies from one part of the registry to another, thereby increasing the possibility of a management error.

Encapsulation. Hierarchical configuration storage, either in the file system or in the registry, paradoxically prevents applications from encapsulating their data in a single location. Settings that relate one application to another must belong to either application, or to a third party. For example, Figure 1 shows the distribution of Microsoft Office 2003 configuration data compared to the overall structure of the registry. As the figure shows, Office stores settings throughout the entire registry. As a result, administrators have difficulty in identifying the settings related to a single application.

Reliability. Configuration data corruption due to aborted or misguided management operations often impacts application and system reliability. For this reason, Windows XP includes a rollback mechanism in its install facility, Windows Vista includes transactional registry access, and Windows System Restore allows the whole registry to be checkpointed. However, these features do not go far enough. Rather, the ability to undo any change at any time is needed to fully support reliable management [4]. Rollback during installation is of little use when failures occur during other management tasks, and external configuration changes may prevent rollback from succeeding. For example, we manually injected errors into the configuration data for Adobe Reader 7.0.8 by deleting settings and found that neither the uninstaller nor the installer would function. In addition, system-wide rollback undoes *all* changes and not just those at fault, forcing the administrator to re-apply updates unnecessarily.

Scoping. Configuration data commonly applies to a fixed scope, such as a single user, an application, or the entire machine. MacOS X [3] and GConf [18], for example, provide a fixed set of scopes with inheritance for propagating settings from a global scope to a per-user scope. However, a fixed set of scopes cannot support common usage scenarios, such as sharing printer or network settings between selected applications. Furthermore, these systems store only a single copy of each scope, so that the users cannot choose at runtime which settings to use. Applications that support switching between groups of settings, such as Firefox's user profiles, must implement the feature themselves. As a result, the semantics of which settings are hidden inside application logic and are not visible to generic management tools.

In summary, we have identified several features where existing configuration storage systems fall short resulting in increased management cost and decreased reliability.

3 Related Work

Much work on configuration management has focused on two aspects of the problem: automating management

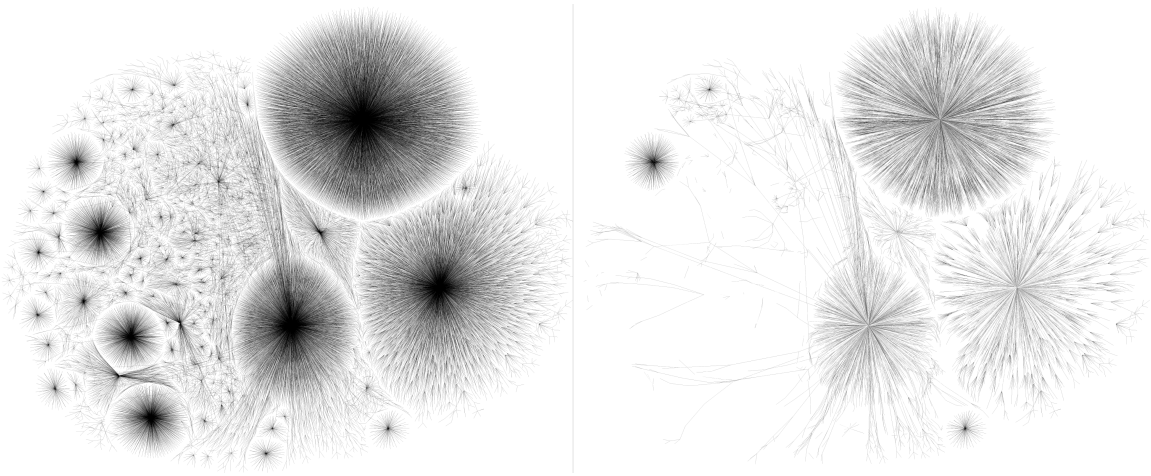

Figure 1: **The graph on the left represents the Windows registry as a whole. Nodes represent registry keys and edges denote a parent-child relationship. The graph on the right includes only registry keys added during Office 2003 installation.**

of large clusters and debugging problems. Automated management systems propagate an operator's configuration changes from a central location to a cluster of machines [5, 2, 1, 15, 16, 13]. This approach works particularly well if the cluster is homogeneous; otherwise the changes must be tailored for each individual machine. Automated management works well until it fails, for example when one machine in a cluster experiences a problem that others do not.

At this point, configuration debugging tools help find the root cause of the problem [22, 23]. These systems examine a set of configuration data, either across multiple machines or on a single machine across time, to determine which changed setting triggered the failure. Here, additional information such as textual descriptions of configuration settings, default values, and data types are useful to aid in understanding how configuration data impacts system behavior.

Finally, there has been recent work proposing wholesale changes to how configuration is performed [7]. This proposal, however, only applies to a subset of configuration data, for example it does not address user preferences or kernel configuration parameters.

Several prior projects have sought to change how configuration data is stored. GConf [18] and Nix [8] both provide new services, although in restricted domains: GConf only applies to user preferences and Nix to package management. In contrast, our work addresses all uses configuration storage.

New models have been proposed recently for configuration management. A purely functional model is proposed in [9]. This model is primarily useful for package management and for generating configuration files. But due to its reliance on text files for storing configuration data, it also suffers from the problems described in previous section. PRESTO [10] constructs configuration settings for devices by composing *configlets*, generated using templates and scripts. However, the use of templates and active template language is more useful for generating a large number of domain-specific configuration settings: configuration files for a large number of machines on a network, for example.

The notion of separating configuration into objects that can be optionally applied is a core feature of Windows group policy objects [17], but these are only used for system settings and not application or user settings. In addition, there is no hierarchy, so only a single object covers a particular setting.

Several aspects of CDMS have been proposed, but not as a single package. Logging configuration changes is one aspect of Flight Data Recorder (FDR) [21]. However, FDR is a full-system tracer, whose overhead may not be appropriate for many cases. Databases have been used for storing configuration data [11], however this approach exported text files and hence could not support applications that modify their own configuration, such as common desktop applications.

4 Configuration Data Model

We propose a new data model for storing configuration data. Existing configuration management services such as the Windows Registry, GConf, and MacOS X property lists store data as a hierarchy of containers storing key-value pairs. Our data model instead provides three key abstractions:

1. *Properties* store individual setting within a hierarchical name space.
2. *Objects* group related properties together, independent of their place in the name space

3. *Spaces* assemble a set of objects into an address space of property names and values visible at runtime.

Figure 2 shows the three key abstractions of our data model and their relationship. We now discuss each in turn.

4.1 Properties

The smallest unit of configuration data is a *property*, which is a binding of a name to a value. Properties form a global namespace, unlike other configuration storage services, which place properties into containers. To achieve a similar hierarchy, we adopt the naming convention of Mozilla Firefox and give each configuration setting a dotted name, leading with a vendor, application, or service name and ending with the name of the individual setting. For example, an Apache setting might be named `apache.v2.timeout`, indicating that the application name is "apache" version 2, and the setting name is "timeout." The left column of Figure 2 shows sample properties.

While simple, this name format supports data already stored in the Registry: key and value names may converted into a dotted name. MacOS X property lists can be stored similarly. To support common operations such as enumeration, the data model provides queries over property names. For example, a list of printers may be specified as `printer.color-floor1`, `printer.bw-floor3`, and may be enumerated by querying for `printer.*`.

Like other configuration systems, this naming mechanism allows applications to group related settings. In addition, it provides a global namespace, which allows a setting to refer to other settings directly. For example, the model supports symbolic links that allow a property to take its value from another property. This enables a new version of an application, with a different property names, to refer to values from previous versions. In contrast, the Windows Registry does not make use of a global name space and instead uses application-specific identifiers, such as GUIDs, to reference other settings. As a result, the relationships between configuration settings are hidden inside application logic and are not visible to administrative tools.

4.2 Configuration Objects

The data model supports two mechanisms for grouping related properties. The first, just described, is to use the name space. Related settings can share a prefix. The second mechanism is *configuration objects*, which are named groups of properties. Each property belongs to a single configuration object. Objects allows settings that are widely dispersed in the namespace to be grouped together and address the problem demonstrated in Figure 1. Despite being dispersed in the namespace, the properties used by Microsoft office can be clustered into a small number of configuration objects. This simplifies uninstallation, as all application-related settings may be removed by removing the associated object. The center column of Figure 2 shows sample configuration objects for different applications and system components.

In addition to grouping related settings, objects also provide a generic mechanism for implementing profiles. Each set of application settings is stored in a separate configuration object and selected when the application starts. An add-on application can store all of its configuration data in a single object, which can then be shared by multiple applications. Any change to the add-on object will then be reflected across all the applications using the add-on.

Objects also provide a mechanism for copying, snapshotting and rolling back settings. Similar to a file, a configuration object may be copied to a new object. This supports periodic snapshots, for undoing changes later found incorrect. It also enables the creation of test configurations to try new settings temporarily. Thus, a property name may exist in multiple configuration objects at the same time, and have a different value in each object. A property is therefore an *instance* of a property name within a configuration object.

4.3 Configuration Spaces

While configuration objects provide a convenient mechanism for grouping properties, they require a mechanism for applications to select *which* objects to use. The data model provides this services with *configuration spaces*, which are named address spaces that map names onto values. This mapping is created by assembling an ordered list of configuration objects. The right column of Figure 2 shows a sample configuration space for the Firefox web browser, composed of Firefox settings and Linux networking and user settings. Settings at the front of the list override settings further away, allowing users or applications to override system settings on a case-by-case basis. Configuration spaces may themselves be nested to simplify management, as a change to one space propagates to all spaces that include it.

These spaces provide better encapsulation of settings than files. All the settings belonging to an application go into a single configuration space, even those that impact system-wide features. Consequently, administrators can quickly find all the settings of an application.

The flat namespace simplifies inheritance, which is performed on a per-name basis: if a property name appears closer to the front of the list, it overrides all instances of the name in more distant objects. By de-

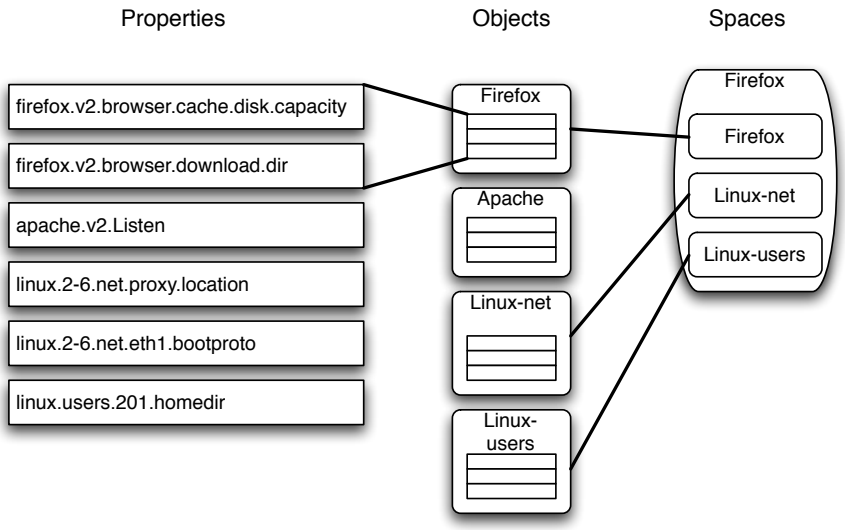

Figure 2: **Data model abstractions.**

fault, all properties may be overridden, allowing per-application versions of system-wide settings. When necessary, inheritance can be disabled for by setting a *mandatory* flag on the property. To enforce non-overridable settings on a finer granularity, an administrator may create a "mandatory settings" object with these settings and place it at the head of the configuration space. An example configuration space for a browser application is shown in Figure 3. The mandatory settings, user preferences and application defaults form an ordered list with the mandatory settings object at the head. To prevent users from changing mandatory settings, the model allows an object to be marked *read-only* in a space.

An aware application may select a configuration space on its own. Or, a launching tool can provide the application with the space to use. This allows a single application to be executed in different spaces, providing the functionality of profiles. Spaces also support executing multiple versions of an application, as settings from these multiple versions exist in different spaces. Furthermore, configuration spaces provide a generic mechanism for sharing settings between selected applications. For example, the printer may default to color for graphics applications and black-and-white for text applications. This provides a finer granularity of control than is available to users today, who typically must choose a single default printer.

In each of these cases, support for different configuration spaces at the system level removes the management and interpretation of configuration from the application. Instead, the service storing configuration data determines which settings are in effect and can communicate that to administrators.

4.4 Extensions

Our configuration data model supports two extensions to simplify management: constraints and metadata.

Constraints. In many cases, applications use internal logic to determine whether an application settings applies. For example, a mail program may store additional settings about external editors if one is enabled. Or, an application may have settings that apply to particular version of the operating system. The application logic applying these constraints makes it difficult for administrators to determine which settings are in effect.

Constraints provide a generic mechanism for conditional configuration data. An application may attach a condition to either a single property or to a configuration object within a space specifying *when* it should apply. Constraints support case-statement functionality: they test whether a property takes on a certain value. If it does, then the property or object is included in the space when an application accesses its settings; otherwise it is not. In the case of spaces, the constraint also specifies where in the space to include the object:

```
if environ.os.name = 'Windows'
      include my_security_object at rank 3
```

The constraint is evaluated at runtime: when accessing data, the property name in a constraint is queried from the current space. This allows the properties used in constraints to themselves be overridden.

Constraints that refer to other properties are called *internal constraints* and can be implemented without application help. The data model also supports *external* constraints, which allow constraints on arbitrary variables. For external constraints, the application must provide the

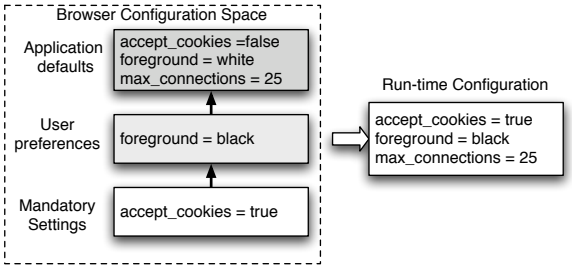

Figure 3: **Configuration objects, Space and Inheritance**

configuration service with a list of variables and their values. When properties are queried, the model evaluates external constraints against the supplied variables. This feature enables control over configuration settings through environment variables and other dynamic application settings, such as the current working directory.

Metadata. One downside of structured configuration storage systems, such as the Windows Registry, is that they do not provide much support for metadata. Comments, default values, and ranges of valid settings are common in text configuration files. Our data model provides explicit space to store such metadata with every property name. This avoids duplicate storage of metadata when a property is overridden. Examples of metadata include comments describing the configuration property, site-specific configuration guidelines, version number, or the user responsible for managing data. This provides many of the benefits of configuration text files while still supporting a richer data model.

5 Configuration Data Management System

In order to get experience with our configuration data model, we implemented the Configuration Data Management System (CDMS), which uses the model. CDMS is a system-wide service intended for storing all configuration data, including system settings, applications settings, and user preferences. This service provides a global view of the configuration data in a system, which facilitates development of shared management tools and interfaces. The centralized system also provides services such as transactions and logging to all applications, avoiding the need for reimplementing these on a per-application basis. In this section we describe the architecture and implementation of CDMS.

5.1 CDMS Architecture

CDMS consists of the three main components shown in Figure 4: the Storage Engine (SE), the Configuration Management Engine (CME) and the user interface layer. The Storage Engine provides the back-end for storing and retrieving configuration data. The Configuration Management Engine implements our data model on top of storage engine as a new query language. Finally, the user interface layer presents a variety of methods for accessing the data, including file import/export, a command line, and an API.

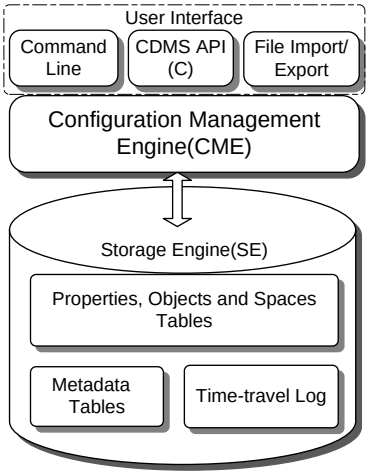

Figure 4: **CDMS Architecture**

The function of the storage engine is to persistently and reliably store all configuration settings. Rather than re-implement much of the functionality already present within databases, we chose to use the PostgreSQL database [19] as our storage engine. It executes as a service to which clients may submit queries. In addition to accessing configuration data, the Storage Engine also provides transactions, to allow request to execute reliably and atomically. Section 5.2 describes the schema for storing configuration data.

The Configuration Management Engine implements our data model on top of storage engine. The main function of CME is to implement a query language for configuration data. We describe the Configuration Management Language, CML, in Section 5.3. The CME translates this language, which refers to properties, objects, and spaces, into queries to the underlying tables provided by the Storage Engine. In addition, the CME logs all changes to configuration data by logging the old value of modified data. The CME provides a time-travel capability for configuration data by searching the log.

The user interface layer provides multiple front ends to CDMS. For applications, there is a C-language API for storing, retrieving, and enumerating properties. For administrators, there is a command-line tool that accepts CML queries (as well as SQL queries for low-level access). Finally, there is a file import/export tool that moves settings between CDMS and files. With knowledge of a file format, a script can be written than generates existing configuration file formats from CDMS data, similar to the PRESTO [10].

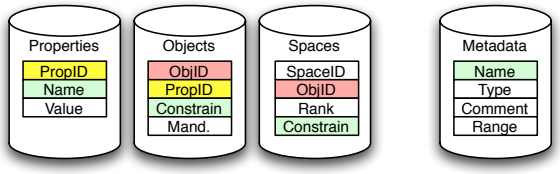

Figure 5: **Tables used to store CDMS data. Fields used for JOINs are colored similarly.**

5.2 Data Storage

CDMS stores all configuration data in relational tables and provides a view of the data as defined by our data model. The three abstractions in our data model - properties, objects and spaces - also manifest in relational schema, as shown in Figure 5. CDMS stores configuration data in *properties*, *objects* and *spaces* tables, created on per-user basis. Each CDMS item has an associated unique identifier (a SpaceID, ObjectID, or PropID) The *spaces* table stores the identifiers of objects composing the space along with the ordering information (a rank). Each of these objects is a collection of properties; the *object* tables stores this mapping from object identifier to properties. The values of the configuration properties are stored in *properties* table. Thus the *properties* table stores name-value pairs, the *objects* table stores (object name,property name) pairs and the *spaces* table stores (space name,object name, rank) triplets, where *rank* represents the order of object in space. In addition, CDMS uses additional fields and tables described below.

Mandatory. The mandatory field of *objects* is similar to Java's *final* keyword and is used to prevent overriding of the setting by other objects in the inheritance hierarchy. This is especially useful for administrators who want to enforce certain settings for an application yet providing them with flexibility to configure others. For example, administrator can defines settings $printer.name = laser$ and $security.allow\_cookies = false$. Further, he may want to prohibit users from overriding the security setting but allow them the freedom to choose printer name. This is achieved by setting mandatory attribute of security property to true.

Constraints. The previous section provided example of a constraint: inclusion of an object in composing space is dependent on the operating system version. CDMS uses a *constraint* field in *objects* and *spaces* tables. An object is included in the space only if the corresponding constraint evaluates to true. Similarly, constraints are used to control inclusion of a property in an object.

Configuration Metadata. CDMS stores configuration metadata, such as the data type of a property and comments on the property in a separate global metadata table. As this data would commonly be repeated in all objects defining a property, separating the metadata ensures that properties have a consistent type and saves space by avoiding repeated storage.

5.3 CML and Query Translation

The Configuration Management Language (CML) is a variant of SQL tailored to configuration data. Rather than allowing access to arbitrary tables, CML provides access to properties, objects, and spaces. The basic CML statements are similar to SQL: SELECT, UPDATE, DELETE and CREATE. Each CML query begins with a statement type followed by a *configuration abstraction* – property, object or space. A simple CML statement is shown below:

```
CML> SELECT OBJECT <object name>
CML>   WHERE PROPERTY = <property name>;
```

The PROPERTY, OBJECT and SPACES terms are treated as keywords. To translate a CML query to SQL, the CME first constructs a parse tree from CML statements. Most CML statement types, for example SELECT, have one-to-one correspondence with SQL and these are translated as such. The SELECT statement is translated to SELECT *. The configuration abstraction following the statement type, provides the relational table name on which the query operates. For SELECT statements, the configuration abstraction name is used to construct FROM clause of SQL. If a configuration abstraction is used in the WHERE clause, it joins the tables. CME uses remaining conditions in WHERE clause as such. The above CML query is translated to following SQL statement:

```
SQL>SELECT * FROM objects o, property p
SQL>  WHERE o.propid = p.propid
SQL>    AND o.name = <object name>
SQL>    AND p.name = <property name>;
```

For the CREATE OBJECT statement, shown below, CDMS first inserts properties in *property* table, SELECT's the identifiers of the inserted properties and creates mapping from object to these identifiers in the *objects* table. CREATE SPACE proceeds similarly. Database transactions allow this to execute atomically.

```
CML> CREATE OBJECT <name> (
CML>   property1 = value1
CML>   property2 = value2
CML>  );
```

The UPDATE and DELETE statements of CML allows addition and deletion of objects and properties, modification to property values, and re-ordering of objects in a space. Executing UPDATE statement proceeds

in three steps: selection of data to be updated, logging the current data values and updating the values. On translating UPDATE statement to SQL, the WHERE clause of generated SQL statements is used to generate SELECT statements. The selected data values, along with timestamp, are INSERTed in the log table. Finally, the UPDATE statements are executed. Examples of UPDATE statements are:

```
CML> UPDATE SPACE <space name>
CML>    DELETE OBJECT <object name>;

CML> UPDATE OBJECT <object name>
CML>   SET PROPERTY <prop. name> = <value>;
```

Thus, CML exposes much of the power of SQL queries to applications and administrators but conceals the underlying implementation of the data model.

5.4 CDMS API

The CDMS API provides functions for applications to read and manipulate configuration data without having to write CML queries. For simple read operations, it abstracts away configuration objects and spaces, and instead provides a simple interface to query property values. The CDMS read functions allows applications to read values of configuration properties, object or a space from their name. Two basic functions read data. The `cdms_read_property` API return a single property value, and is useful for applications that load configuration data on demand. The `cdms_enum_properties` API returns all properties with the provided prefix. This is useful for applications that load all their settings on startup, to enumerate all settings, and to access variable-sized lists. For example, an application may use this API to retrieve a list of the available printers or fonts.

On a read query, the API constructs a CML query and submits it to the CME for execution. Thus, these APIs returns only the property values that "win" the application of inheritance. The CDMS API does not allows applications to read values that are not active.

The CDMS API for writing properties, `cdms_write_property`, takes an optional *object* parameter to specify which objects hold receive the new value. If not present, the object defaults to the object containing the current value of the property, if writable, and if not, the first writable object in the space (there may be read-only objects at the front to enforce mandatory settings).

Beyond property access, the CDMS API include additional functions to create and delete objects, and spaces. These are largely intended for sophisticated applications To create an object, CDMS provides the `cdms_create_object` function that takes object name and a list of key-value pairs as parameters. To create a space, the `cdms_create_space` takes a space name and a list of object names in rank order.

Finally, the CDMS API has functions to read constraints on an object or property. We are currently implementing support for evaluating internal and external constraints. Once developed, the API will include a function call that takes variable values as inputs and evaluates constraints on behalf of applications.

5.5 Assembling Configuration Space

Assembling configuration space of an application is one of the most common tasks for CDMS, and the most important one too. CDMS assembles a space in response to query SELECT SPACE <name> or in response to function call `cdms_read_*`. The data to map a space name to the properties it contains is stored across two layers of indirection: the *spaces* table stores mapping from space name to objects and the *objects* table maps these objects to *properties*. The *spaces* table also stores ordering of objects in space using *rank* field.

To query values in a space, CME issues a SQL query that first joins the three tables, *spaces*, *objects* and *properties* to map space name to properties. These tuples are then ordered by property name and object rank in the space and returned to CME. The tuples returned by this query may contain duplicate properties if multiple objects defining the property. CME applies two inheritance rules to select the correct value: (1) the lowest ranked mandatory setting is selected and if no setting is marked as mandatory, then (2) the setting with highest rank is selected. While we implement this functionality in CME, databases that support the WINDOW clause could implement this as part of query processing [20].

CDMS supports two mechanisms for associating an application request with a space. First, the application may explicitly specify the space name. This is appropriate for sophisticated applications that manage their own spaces. Alternatively, we have written a *launcher* that specifies the space name for an application in an environment variable. The CDMS API library retrieves this environment variable when making queries. The launcher stores it association of application names and spaces within CDMS, and normally runs in the default space of the user. However, because its data is in CDMS, the association of spaces and application can be overridden using inheritance.

5.6 CDMS Services

The core features of the CDMS storage and data model can serve as the foundation for additional functionality.

Profiles. Some applications support multiple groups of preferences and allow a user to choose a group to use, for example based on his current project. Profiles are easily provided by CDMS with configuration objects. Each

profile stores the settings unique to the profile in a single object, while settings common to all profiles are stored in a separate object. To use a profile, the user constructs a configuration space with the per-profile object as the head followed by the common object. The same feature can be used for system settings, such as adapting network settings to different environments.

Time Travel. CDMS can provide time travel either through its persistent log or by snapshotting configuration objects. With the log, a user or administrator can roll back the changes to all objects, any single object, or just a specific change. With snapshots, time-travel is implemented by starting an application in a configuration space using the snapshot.

Multiple Versions. Multiple versions of an application or service can coexist because their settings are stored in distinct configuration objects. Installation of a new version will not overwrite the configuration settings of older version. A user or administrator can then select the desired version by constructing a space referring to that version.

5.7 Summary

CDMS improves simplifies management by grouping settings into configuration objects that may be organized into a configuration space. CDMS moves the semantics of which settings are in effect out of application logic and into the inheritance mechanism, which exposes the configuration of applications to administrators. In addition, CDMS stores data in a database, which supports transactions and logging, which improve reliability by preventing partial updates and enabling partial rollback.

6 Case Studies

We illustrate how CDMS stores and manages configuration data with the examples of Firefox web browser and Apache web server. We have little operational experience with CDMS at this point, so we focus instead on how CDMS can store configuration data and support flexible management. Firefox is a naturally customer for administrative controls and inheritance, and we demonstrate how to apply these features. Apache has a wide variety of irregular configuration data, including both system-wide settings and per-directory settings. We illustrate how CDMS supports this complex data.

6.1 Configuring Firefox

We take as an example the problem of configuring Firefox web browser in an academic network. Firefox already stores it configuration using a dotted-name convention in a file named "prefs.js". This data can be directly stored and accessed in CDMS without any need for reformatting the data.

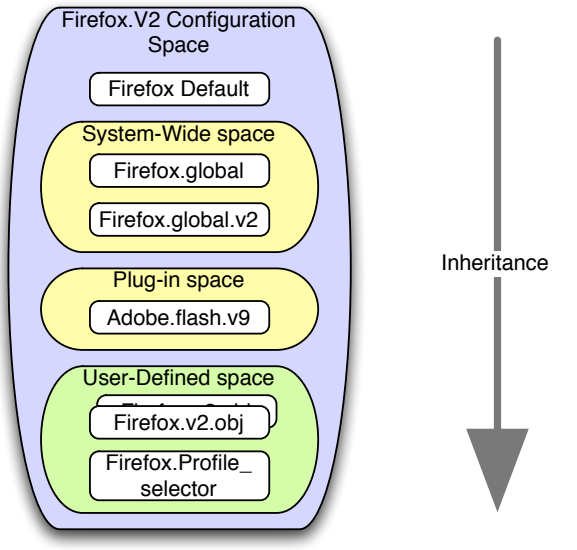

Figure 6: **Firefox configuration data in CDMS. White rectangles represent configuration objects, and shaded shapes are configuration spaces.**

We consider a version of Firefox modified to use CDMS (although we have not constructed one). An administrator is responsible for installing applications and configuring site-specific policies, such as whether passwords should be stored. The network has two primary categories of users: faculty and students. An example Firefox configuration space is show in figure 6. We next discuss the how the components of this space support the application, administrator, and user.

Application Installation. During installation, the application uploads its defaults settings into the `Firefox Default` configuration object. At the time of installation, administrators specifies a unique name for the configuration space, `Firefox.V2`. The space includes default settings, administrator-specified system-wide settings, and user-defined settings. The installation process updates the table used by the launcher to associate Firefox with the `Firefox.V2` space.

Administrator's Role. An administrator customizes Firefox application by creating an object in the system-wide Firefox configuration space. The administrator may define some of the settings as mandatory, as described in previous section, which prevents users from overriding these settings. The inheritance mechanism of our model facilitates more flexible customization. For example, consider a system with an old version of Firefox that stores settings in the *Firefox.global* object. Settings from older version that are also applicable to the new version are simply inherited by adding the *Firefox.global* object to the configuration space as shown in Figure 6. This also simplifies management: administrators are able to cus-

tomize multiple versions by just modifying the settings in `Firefox.global` object. Customizing only one version is also simple: modify `Firefox.global.v2`, which is specific to version 2.

Configuration data of add-ons and plug-ins resides in its own space (`plug-in`) that is included in Firefox configuration space. Similar to sharing the of configuration object by multiple versions, CDMS also facilitates sharing of configuration objects of add-ons and plug-ins by multiple applications(or by multiple versions of an application).

User's Role. The configuration space of every application always contains a default object and system-wide settings, as shown in Figure 6. To override default settings, user creates a configuration object and adds it to the user-defined `Firefox.v2` configuration space. CDMS also allows the user to create profiles, which are multiple versions of the user-defined configuration space. CDMS selects appropriate profile at run-time based on the an settings in a special `profile_selector`, which is set by the launcher.

Application View. Firefox can use the CDMS API to enumerate all the properties when starting or read selected properties when needed. When the user changes a setting through the user interface, CDMS writes to the first writable object, which will be the `Firefox.v2.obj` object. The other objects (environment and selector) are not writable by the application. As a result, much of the logic related to configuration data is moved out of the Firefox and into CDMS.

Uninstallation. CDMS facilitates clean uninstallation of the application. The properties associated with Firefox are all stored in distinct configuration objects (`Firefox.global`, `Firefox.global.v2`, `Firefox.v2.obj`). Removing these objects removes all the Firefox-related settings. Other objects, like add-on configuration objects, are retained if they are used in other configuration spaces; otherwise they could be garbage collected.

6.2 Configuring Apache Web Server

Unlike Firefox, which already stores configuration settings as a list of properties with dotted names, Apache configuration files exhibit more complex structure. Many settings are simple name–value pairs, such as the root directory. However, Apache also stores per-directory settings and conditional settings. A snippet of Apache's configuration file is shown below:

```
ServerRoot "/etc/httpd"
<IfDefine SSL>
      Listen 80
      Listen 443
</IfDefine>
<Directory "/scratch/apache/htdocs">
    Options Indexes FollowSymLinks
    Order allow,deny
    Allow from all
</Directory>
```

In the above example, multiple configuration directives are placed under the `<Directory>` directive. The data is not in "flat" or key-value format. The data has hierarchical structure coupled with a constraint on the applicability of enclosed attributes. This hierarchical structure must be maintained when storing the data in CDMS, which stores a flat list of name–value pairs.

Table 1 shows line by line a possible mapping of Apache settings to CDMS properties. The configuration settings are all prefixed with a unique identifier, `apache.v2`, to distinguish the application name and version. Apache loads all settings when starting, using `cdms_enum_properties` API to retrieve all properties prefixed with `apache.v2`. Simple settings, such as the root directory, can be stored directly, for example as `apache.v2.ServerRoot`.

Conditional settings rely on the constraint feature. For example, the constraint for `Listen` is:

```
apache.v2.SSL = 'TRUE'
```

To store multiple values of Listen, a unique number must be appended: `apache.v2.listen.1` and `apache.v2.listen.2`.

Settings enclosed within the `<Directory>` directive are flattened by adding name `Directory.x`, where `x` is a number unique to each directory. This maintains the the existing naming hierarchy.

As previously described, much of the contents of Apache's configuration file are comments. These comments contain both advice to administrators and disabled settings. CDMS stores the advice as metadata comments on a property. The disabled settings are stored as properties with "null" values, which do not show up when queried but exist as placeholders.

7 What About Files?

Any new configuration service must compete against the simplicity, flexibility, and entrenchment of text files. In this section, we describe how CDMS can achieve many of the same benefits of text files in addition to providing valuable new services.

Copying. Copying and renaming of files enables alternate configurations, selective backup, and passing configurations to other users or systems. CDMS can provide alternate configurations and selective backup by duplicating configuration objects. These new objects can serve as a backup or as configuration for other users and applications. An additional benefit is that application can be

httpd.conf	Properties Table		
	id	property name	property value
ServerRoot “/etc/httpd”	1	apache.v2.ServerRoot	/etc/httpd
<IfDefine SSL> Listen 80	2	apache.v2.Listen.1	80 **constraint** apache.v2.SSL = ’TRUE’
Listen 443 </IfDefine>	3	apache.v2.Listen.2	443 **constraint** apache.v2.SSL = ’TRUE’
<Directory ”/scratch/apache/htdocs”>	4	apache.v2.Directory.1.**constraint**	= /scratch/apache/htdocs
Options Indexes FollowSymLinks	5	apache.v2.Directory.1.Options	Indexes FollowSymLinks
Order allow,deny </Directory>	6	apache.v2.Directory.1.Order	allow,deny

Table 1: **Importing Apache Configuration Data in CDMS. We show internal constraints in the properties column for clarity, although their are stored in the objects table.**

configured to use *any* version of its settings through configuration spaces. In contrast, with files the single configuration file must be changed to select a group of settings. CDMS can transmit data between systems by exporting a configuration object to a file, which can then be imported on other systems.

Search. Administrators often use *grep* to search for settings. Within CDMS, administrators may issue CML queries against property names and values and metadata. Because CDMS centralizes settings, an administrator need only search a single location as compared to all the directories that may contain configuration files. Furthermore, the additional semantics attached to data, can reduce false positives by distinguishing metadata from property values and names.

Encapsulation. Configuration files encapsulate the majority of the configuration state of an application and act as a location for managing its settings. This same encapsulation can be provided in CDMS through two complementary mechanisms: naming and configuration objects. The naming convention causes the private configuration state of an application to be contained under a single name prefix, allowing it all to be easily examined. Configuration objects contain both the private and shared state of an object, providing better encapsulation than files, where shared state resides in separate files.

Metadata. Text files, by their flexibility, simplify the addition of comments and other metadata, including default values, to configuration files. For example, 683 of the 940 lines in the default `httpd.conf` file for the Apache web server are comments. Many of these comments are optional values, sample settings, and descriptions. CDMS can support this information as metadata, attached to properties. Optional settings, indicated with commented out values in a configuration file, are instead stored as properties with “null” values.

Tool support. Administrators often write scripts that read or manipulate configuration text files. This provides a simple mechanism to automate management tasks. A similar level of programmability is offered through the CML query language, which offers the ability to manipulate configuration data directly. For example, a script can invoke a CML command to propagate new settings, check for changes, or return current applications versions.

Compared to file-based configuration storage, CDMS can reduce system administration costs, provide extensive functionality, simplify application development, and improve system reliability.

8 Conclusion

We believe that the current systems for storing configuration data are hampering system management by providing abstractions that are far below what applications desire. As a result, applications layer their own semantics upon the service, concealing the true state of the system behind a thick layer of application logic. This is evidenced by the numerous file formats in Linux and the complex graph of data used by Microsoft Office on Windows.

To address this problem, we propose a new model for storing configuration data based on three abstractions: properties, objects, and spaces. These abstractions allow flexible management of configuration data and move much of the logic for accessing data out of applications and into a common service, where it can be accessed by system managers. This simplifies management, because the model exposes the *exact set* of properties enabled by an instance of an application. In contrast, current configuration services provide little assistance to administrators to determine which settings are in use by an application.

We implemented this data model in CDMS, the Con-

figuration Data Management Service. The service offers additional benefits to applications and administrators: transactions for reliability, logging for rollback and inspection of changes over time, and a query language to support rich management tools. In addition, it provides many of the benefits offered by simple text files, such as copying data, searching, and rich metadata. We plan to convert several applications to use CDMS, both natively and through file import/export, to gain more operational experience using it as a management tool.

Acknowledgement We would like to thank Matthew Renzelmann for providing understanding Windows Registry and for generating Figure 1.

References

[1] P. Anderson. Towards a high-level machine configuration system. In *8th USENIX LISA*, 1993.

[2] P. Anderson and A. Scobie. LCFG - the Next Generation. In *UKUUG Winter Conference*. UKUUG, 2002.

[3] Apple Inc. Runtime configuration guidelines. `http://developer.apple.com/documentation/MacOSX/Conceptual/BPRuntimeConfig/BPRuntimeConfig.pdf`, 2006.

[4] A. Brown. Toward system-wide undo for distributed services. Technical Report UCB/CSD-03-1298, EECS Department, University of California, Berkeley, 2003.

[5] M. Burgess. Cfengine: A site configuration engine. *USENIX Computing systems*, 8(3), 1995.

[6] Computing Research Association. Final report of the cra conference on grand research challenges in information systems. `http://www.cra.org/reports/gc.systems.pdf`, 2003.

[7] J. DeTreville. Making system configuration more declarative. In *10th USENIX HotOS*. June 2005.

[8] E. Dolstra, M. de Jonge, and E. Visser. Nix: A safe and policy-free system for software deployment. In *18th USENIX LISA*, 2004.

[9] E. Dolstra and A. Hemel. Purely functional system configuration management. In *11th USENIX HotOS*, May 2007.

[10] W. Enck, P. McDaniel, S. Sen, P. Sebos, S. Spoerel, A. Greenberg, S. Rao, and W. Aiello. Configuration management at massive scale: System design and experience. In *2007 USENIX ATC*, June 2007.

[11] J. Finke. An improved approach for generating configuration files from a database. In *14th USENIX LISA*, 2000.

[12] A. Ganapathi, Y.-M. Wang, N. Lao, and J.-R. Wen. Why pcs are fragile and what we can do about it: A study of windows registry problems. In *2004 IEEE DSN*, 2004.

[13] Hewlett-Packard Corp. HP OpenView. `http://h20229.www2.hp.com/`.

[14] E. Kiciman and Y.-M. Wang. Discovering correctness constraints for self-management of system configuration. In *1st Intl. Conf. on Autonomic Computing (ICAC)*, 2004.

[15] W. LeFebvre and D. Snyder. Auto-configuration by file construction: Configuration management with newfig. In *18th USENIX LISA*, 2004.

[16] Microsoft Corp. Microsoft systems management server. `http://www.microsoft.com/smserver/default.mspx`.

[17] Microsoft Corp. Windows server 2003 group policy. `http://technet2.microsoft.com/windowsserver/en/technologies/featured/gp/default.mspx`.

[18] H. Pennington. Gconf: Manageable user preferences. In *2002 Ottawa Linux Symp.*, June 2002.

[19] PostgreSQL Global Development Group. PostgreSQL home page. `ttp://www.postgresql.org`.

[20] SQL Anywhere Server. SQL reference – WINDOW clause. `http://www.ianywhere.com/developer/product_manuals/sqlanywhere/1000/en/html/dbrfen10/rf-window-clause-statement.html`.

[21] C. Verbowski, E. Kiciman, A. Kumar, B. Daniels, S. Lu, J. Lee, Y.-M. Wang, and R. Roussev. Flight data recorder: Monitoring persistent-state interactions to improve systems management. In *7th USENIX OSDI*, Nov. 2006.

[22] Y.-M. Wang, C. Verbowski, J. Dunagan, Y. Chen, H. J. Wang, C. Yuan, and Z. Zhang. Strider: A black-box, state-based approach to change and configuration management and support. In *17th USENIX LISA*, 2003.

[23] A. Whitaker, R. S. Cox, and S. D. Gribble. Configuration debugging as search: Finding the needle in the haystack. In *6th USENIX OSDI*, Dec. 2004.

[24] J. Xu, Z. Kalbarczyk, and R. K. Iyer. Networked windows nt system field failure data analysis. In *1999 Pacific Rim Intl. Symp. on Dependable Computing*, Dec. 1999.