Wallaby: A Scalable Semantic Configuration Service for Grids and Clouds

William C. Benton
Red Hat, Inc.
willb@redhat.com

Robert H. Rati
Red Hat, Inc.
rrati@redhat.com

Erik J. Erlandson
Red Hat, Inc.
eerlands@redhat.com

Abstract

Job schedulers for grids and clouds can offer great generality and configurability, but they typically do so at the cost of increased administrator complexity. In this paper, we present *Wallaby*, an open-source, scalable configuration service for compute resources managed by the Condor high-throughput computing system. Wallaby offers several notable advantages over similar systems: it lets administrators write declarative specifications of user-visible functionality on groups of nodes instead of low-level configuration file fragments; it presents a high-level semantic model of Condor features and their interactions and dependencies; it validates configurations before pushing them to nodes; it supports version control, "undo," and configuration differencing; and it includes a networked API that enables extensions and advanced functionality. Wallaby allows administrators to extend pools to include more physical, virtual, or cloud nodes with minimal explicit configuration. Finally, it is scalable, supporting pools consisting of thousands of nodes with hundreds of configuration parameters each.

1. Introduction

Distributed schedulers typically offer many configurable parameters, ranging from hostnames of machines running system components to security mechanisms, and from performance-tuning options to definitions for job-scheduling policies. The Condor high throughput computing system [5] is no exception: it is enormously flexible, but the depth and scope of its runtime configurability can be daunting to compute center administrators. In the recent 7.6 release series of the Condor system, the bundled "generic" configuration file[1] is 2,651 lines long and contains 592 parameter assignments. Many of these assignments are commented out in order to serve as documentation for users who might wish to enable particular functionality, but many others are required for proper operation of the system.

[1] This is `src/condor_examples/condor_config.generic` in the Condor source repository.

SC'11, November 12–18, 2011, Seattle, Washington, USA.
ACM 978-1-4503-0771-0/11/11

Some of the difficulties of configuring distributed applications are well-understood and even addressed in widely-used systems: synthesizing configurations from multiple configuration snippets or imperative recipes, notifying nodes of updated configurations, efficiently pushing updated configuration files to many nodes, and perhaps providing some rollback functionality in the event of a spurious configuration. However, no widely-used configuration management system addresses one of the core difficulties in configuring *any* application: managing the complexity of many potentially interdependent low-level parameters in order to achieve a coherent configuration enabling the administrator's desired application-level functionality.

In this paper, we present Wallaby, a configuration service for Condor pools that was designed to solve both the mechanical problems and the semantic problems of distributed configuration. By "mechanical problems," we mean that Wallaby supports synthesizing node configurations from modular, independent partial configurations; it efficiently notifies nodes of changes to their configurations and deploys updated configuration files as necessary; and it keeps all configurations under an easy-to-use version control system that enables rollback or "undo" of configurations as well as differencing of deployed and updated configurations. The "semantic problems" that Wallaby solves are those related to actually ensuring that a configuration is sensible and functional before it is deployed to nodes: specifically, it is based on a semantic model of Condor configurations and can validate that a configuration satisfies an expressive and useful set of constraints before deploying it.

The capacity to model properties of valid and invalid Condor configurations is a necessary but not sufficient condition for solving the semantic problems of distributed configuration. Wallaby also includes a comprehensive database of semantic metadata for Condor parameters and interesting user-visible pieces of functionality. By incorporating this database, developed by domain experts, Wallaby enables even administrators who are Condor novices to develop sophisticated and correct configurations quickly and easily.

Wallaby is a mature, real-world system that anyone can download and use today. It has been a publicly-available open-source project, released under the permissive Apache license, for two years; its first public release was in December 2009. Red Hat has shipped a supported version of Wallaby as the preferred way to manage Condor configurations in the last two releases of the Red Hat Enterprise MRG product.[2]

[2] Red Hat Enterprise MRG is Red Hat's Messaging, Realtime, and Grid offering; the first two releases to include Wallaby were Version 1.3, in October 2010, and version 2.0, in June 2011.

1.1 Overview of contributions

In this paper, we present the following contributions:

- A semantic model for characterizing and validating distributed application configurations,
- The Wallaby service, an efficient versioned configuration store for Condor pools that supports declarative configuration specifications that are validated against a semantic model,
- An expressive network-accessible programming interface to enable users to develop new configuration, inventory, testing, and management tools using Wallaby as infrastructure, and
- Two novel methods for evaluating the scale of configuration services.

The remainder of this paper includes background information on the details of the Condor system that will be helpful to understand Wallaby; an overview of the architecture of the Wallaby system; details on the semantic model of configuration and of the algorithms we use to check configuration consistency; and a brief, broad presentation of the Wallaby API and some notable API clients. We conclude by presenting some observations from our experience using and supporting Wallaby over the last two years and placing our contributions in the context of related work.

2. Background

We begin by briefly reviewing some of the details of the well-known Condor system that are relevant to our presentation of Wallaby. This presentation is intended to make this paper self-contained and is by no means exhaustive; for more details on the architecture of Condor, please see chapters 2–4 of the Condor manual [6].

Condor is essentially a distributed scheduler; at a high level, its purpose is to match resources with requests for resources. It does this by matching a *classified advertisement* (ClassAd) expression representing a resource with one matching a request. These expressions can express both *requirements* and *preferences*: for example, a ClassAd expression for a request might specify that a job must run on an x86 Linux machine with at least 8 GB of RAM or that a job would prefer to run on a machine owned by the job owner's research group. It is important to note that an attempted match may fail because a resource rejects a request or because a request rejects a potential resource.

The most important machine in a Condor pool is the *central manager*, which is a machine running the Condor `collector` and `negotiator` processes. These maintain a live store of resource information and match requests to resources, respectively. There will only be one active central manager in a pool at any time (although there may be several passive replicas in high-availability settings). Condor *submit nodes* are those that maintain a queue of submitted jobs in a Condor `schedd` process, attempt to negotiate matches for these jobs, and follow the activity of submitted jobs via shadow processes. Finally, Condor *execute nodes* are those that run a Condor `startd` process; these are advertised to the Condor `collector` as resources.

Every Condor process runs under a watchdog daemon called the Condor `master`. The `master` is responsible for monitoring child processes and restarting them if they crash. Condor is sufficiently flexible to allow administrators to run other processes, including custom Condor components, under the Condor `master`; as we shall see in the next section, Wallaby uses this capability to run its configuration daemon on each machine in a Condor pool.

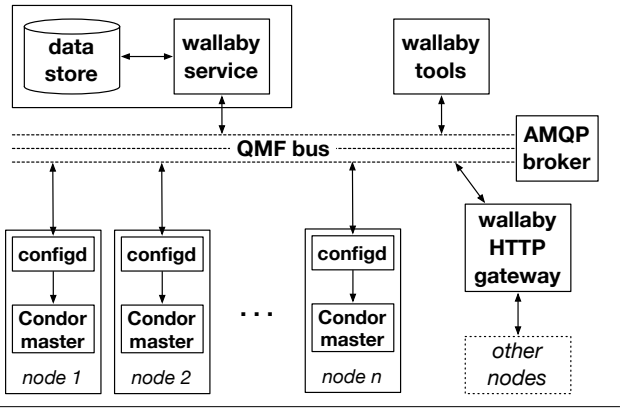

Figure 1. Diagram of a Wallaby-managed Condor pool

3. System architecture

Wallaby has several main components: a networked service with an object-oriented API, a configuration daemon that runs on each Condor node, and a collection of configuration tools that run against the networked API. The networked API uses the QMF managed objects framework running on an AMQP messaging broker (see Vinoski [8] for an early overview of the goals behind the AMQP project).[3] Because the Wallaby API uses AMQP as a transport, clients can both make synchronous remote calls and receive asynchronous notifications, for example, when a node's configuration is updated.

Figure 1 presents an overview of how these pieces fit together. We shall now discuss each piece by presenting an administrator's typical interaction with the Wallaby system. Our running example scenario consists of enabling Condor's power-management functionality for all of the nodes in a particular rack of machines, which the administrator has already placed in a group designated "On-demand workers."

Because Wallaby uses a high-level, semantic description of Condor configurations, the main user interaction in this scenario will be altering the configuration of the "On-demand workers" group, replacing the "Execute Node" feature, which models a standard compute node, with the "Power-Managed Node" feature, which includes hibernation and wakeup policies. In order to do this, the administrator will use a tool built on the Wallaby API.

The configuration tool communicates with the Wallaby service and updates the working state of the service to reflect changes; however, it does not automatically validate configurations according to the semantic model or push updated configuration files out to affected nodes without user approval. This enables administrators to make several changes at once without pushing out a new configuration. Once the administrator is happy with his or her changes, Wallaby validates the configuration (using the algorithms we describe in Section 4) and broadcasts a notification that updated configurations are available to affected nodes. Because only the "On-demand workers" group has changed

[3] Typically, the Wallaby service will run on the same machine as the AMQP broker or the Condor central manager, but this is not strictly necessary.

in our example, Wallaby only validates and notifies the nodes in this group. Configuration validation may fail for several reasons, which we discuss in Section 5.2; in the event that it does, the Wallaby API returns a detailed, descriptive error message describing the cause of the failure.

The configuration daemon is a process running on each Condor node under the Condor master daemon. It responds to notifications over the QMF bus and also checks in periodically with the Wallaby service; as a result, missed notifications delay adoption of a new configuration for a short time but do not affect correctness. Once the configuration daemon discovers a new configuration for its node, it checks in with Wallaby in order to get the contents of the updated configuration and a list of affected subsystems, or Condor processes that will need to be reconfigured or restarted as a result of the difference between the configuration the node is currently running and the new configuration.

The configuration daemon then augments the supplied configuration in two ways:

1. It ensures that the Condor master daemon is configured to run on the node, and that the configuration daemon is also set to run. This ensures that an administrator cannot accidentally disable Wallaby control of a node, which would require manual intervention to fix; and
2. It adds configuration entries so the node advertises the names of the Wallaby features it has installed and the Wallaby groups it is a member of, so that these values may be used in Condor matchmaking.

Finally, the configuration daemon installs the new configuration files and restarts the affected Condor subsystems.

Figure 1 also mentions an alternative read-only interface to configuration data for machines that cannot access the same AMQP broker as the Wallaby service. The Wallaby HTTP gateway is a Wallaby API client that presents a read-only web service interface to configuration data. It can be used with naïve configuration clients that pull down configurations via HTTP GET.

We will now discuss the semantic model of configuration Wallaby presents to pool administrators and employs for configuration validation.

4. Semantic configuration

Wallaby supports *semantic configuration* of Condor functionality and policy. By this, we mean that Wallaby users deal with high-level entities, like user-visible features, rather than with low-level configuration-file macros and parameter settings. This approach has several benefits:

1. Individual user-visible features and policies can be defined by Condor experts, insulating pool administrators from the details of configuration files;
2. A semantic model of configuration allows Wallaby to perform *static configuration validation*, both at a high semantic level and at the level of generated configuration files: Wallaby can verify that each node's configuration fulfills dependencies between features and parameters, avoid installing conflicting features, and provide meaningful, user-friendly feedback immediately after attempted configuration changes that would introduce errors; and
3. Wallaby-managed nodes can automatically advertise the high-level features they offer or their group memberships, so that Condor jobs can choose to match against nodes that have certain features enabled, or have been placed in certain groups.

4.1 Nodes

The Wallaby semantic model of Condor configurations begins with *nodes*. A node corresponds to a single physical or virtual machine running the Condor master daemon on a given host. Wallaby can configure a node to be partitioned into multiple Condor-managed resources, but a node can have at most one Wallaby configuration at a time. However, Wallaby does not support directly applying configurations to nodes.

4.2 Groups

The basic configurable entity in Wallaby's semantic model is the *group*. Nodes can be members of arbitrarily many groups, and these memberships are specified in priority order for each node. In addition to *explicit groups*, whose membership is specified as an explicit set of nodes, there are two kinds of special groups: a single *default group*, which is the lowest-priority membership for every node, and one *identity group* for each node, which only contains that node and is its highest priority membership. Because of these special groups, the act of applying a configuration to a group is fully general. That is, applying a configuration to the default group can enable pool-wide functionality; applying a configuration to an identity group enables it only on a single node. A group's configuration consists of an ordered sequence of features and an optional set of parameter-value mappings.

4.3 Features

A *feature* models a user-visible piece of functionality or part of a policy specification. At its simplest, a feature contains a set of parameter-value mappings that could be directly inserted into a Condor configuration file; in this case, it merely supplies high-level documentation for configuration settings. However, the feature concept is substantially more powerful than a mere configuration file snippet; this power comes from the relationships between features that Wallaby models. Features may *include* other features, they may *depend upon* other features, and they may *conflict with* other features.

Feature inclusion models an extension relationship; it is analogous to implementation inheritance in object-oriented programming languages. Inclusion allows a feature to specialize its included features by adding configuration parameters to their configurations or overriding their settings. More concretely, we can define the configuration $C(F)$ for a given feature F inductively as follows:

1. If F includes features $F_1, \cdots, F_n$, from highest to lowest priority, then $C(F)$ will include $C(F_1), \cdots, C(F_n)$. In the event that $C(F_i)$ and $C(F_j)$ both define a value for the same parameter, the definition from the feature with the higher priority will take precedence.
2. $C(F)$ includes all of the parameter-value mappings *immediately defined* in F, with any immediately-defined parameter-value mapping taking precedence over mappings for the same parameter from an included feature's configuration.

The other two kinds of feature relationships are *dependences* and *conflicts*. If some feature F depends on F_d, Wallaby's configuration validation will ensure that any node that has F installed will also have F_d installed. Likewise, if F conflicts with F_c, Wallaby will ensure that any node that has F installed does not also have F_c installed. The

relation defined by feature dependency, like that defined by inclusion, is acyclic. The relation defined by feature conflict is symmetric by definition; that is, it is sufficient to have one of F conflicts-with F_c and F_c conflicts-with F, because either will prevent both F and F_c from being installed on the same node.

4.4 Parameters

Wallaby also models low-level Condor configuration parameters, in order to ensure that they are given acceptable values, to establish configuration-wide consistency properties, and to provide user-level documentation about low-level settings. For each parameter, Wallaby tracks a *type*, a *documentation string*, an optional default value, sets of parameter dependences and conflicts, a flag indicating whether or not Condor processes must be restarted for a change in this parameter to take effect, and a flag indicating whether or not the parameter must be changed from its default value. This flag is useful for modeling parameters that must be set to provide basic functionality, e.g., the hostnames of machines running important services.

4.5 Subsystems

A *subsystem* corresponds to a Condor application process. Wallaby models subsystems whose operation may be affected by a configuration change in order to either restart these processes or tell them to reread their configuration files. The model of a subsystem consists of a unique name and a set of parameters of interest. Since Wallaby models whether or not parameter-value changes are visible without a restart, it can notify nodes which processes need to reload their configurations and which need to restart after a configuration change.

5. The configuration algorithms

Now that we have discussed the semantic model of configurations, we can proceed to a more concrete view of how Wallaby computes and validates configurations. We will begin by presenting a basic version of the configuration generation algorithm, which answers the question "what is the current configuration for this node?" We will then discuss the properties that valid configurations must satisfy. Finally, we will present improvements, as implemented in Wallaby, to the expressivity and performance of the basic techniques.

5.1 Configuration generation algorithms

The basic idea behind Wallaby's configuration generation algorithm is that a node's configuration is derived from the groups it is a member of, and a group's configuration is derived from the features it has installed.

We define a node's configuration by taking the configurations for each group that it is a member of and merging these together so that that conflicting bindings from different groups take precedence in membership priority order (with the default group at lowest precedence and the node's identity group at highest precedence, as special cases). The configuration for a group is similarly defined as its explicit parameter settings merged over the configurations for each feature it has installed, with conflicts again resolved in priority order. We will briefly defer discussion of how feature configurations are calculated in order to examine the node-configuration algorithm in more detail.

Figure 3 presents get-node-config, a procedure to calculate the configuration for a node stored in *node*; we summarize the interfaces and behaviors of auxiliary procedures in Figure 2. Line 1 initializes the configuration for *node* to an empty dictionary. In lines 2–15, the procedure repeatedly applies the configuration of each group that *node* is a member of to *config*. Note that the algorithm iterates over the reverse of the memberships list in order to apply in inverse priority order, so that higher-priority settings take precedence over lower-priority ones.

Lines 4–8 and 9–11 calculate the configuration for a group by repeatedly applying the configuration for each feature installed on that group before merging in that group's explicit parameter settings. Note in particular the apply procedure, which is defined in Figure 4; in this simple version of the algorithm, it will merely update the configuration, replacing preexisting bindings if they appear. We will discuss extensions to the apply procedure that enable more expressive configurations later.

The algorithm for calculating a feature's configuration is given in Figure 5. Wallaby calculates the configuration for a feature by recursively merging the configurations of the

Name	Description
new-dictionary()	returns a newly-allocated empty dictionary structure
reverse(l)	returns the reverse of list l
memberships(n)	returns a list of the group memberships for node n, from highest to lowest priority
features-for-group(g)	returns a list of the features installed on group g, from highest to lowest priority
params-for-group(g)	returns the explicit parameter-value mapping installed on group g
features-included-by(f)	returns a list of the features included by feature f, from highest to lowest priority
params-for-feature(f)	returns the explicit parameter-value mapping immediately defined by feature f

Figure 2. Auxiliary procedures called by configuration-generation algorithms

```
1:  config ← new-dictionary()
2:  for group in reverse(memberships(node)) do
3:      groupConfig ← new-dictionary()
4:      for feature in reverse(features-for-group(group)) do
5:          for p, v in get-feature-config(feature) do
6:              groupConfig ← apply(groupConfig, p, v)
7:          end
8:      end
9:      for p, v in params-for-group(group) do
10:         groupConfig ← apply(groupConfig, p, v)
11:     end
12:     for param, value in groupConfig do
13:         config ← apply(config, param, value)
14:     end
15: end
16: return config
```

Figure 3. get-node-config, the procedure to calculate the configuration for a node

```
1: config[p] ← v
2: return config
```

Figure 4. A simplified apply procedure, which introduces a parameter-value mapping from $p \to v$ to *config*.

```
1: config ← new-dictionary()
2: for f in reverse(features-included-by(feature)) do
3:     for p, v in get-feature-config(f) do
4:         config ← apply(config, param, value)
5:     end
6:     for p, v in params-for-feature(f) do
7:         config ← apply(config, p, v)
8:     end
9: end
10: return config
```

Figure 5. get-feature-config, the procedure to calculate the configuration for a feature

features it includes in inverse priority order (analogously to the configuration for nodes), and then merging the parameter-value mappings immediately defined by the feature on top of these.

Given these procedures, we can calculate the configuration for each node in a pool. However, these procedures do not guarantee that a configuration is valid given Wallaby's semantic model. In fact, a particular kind of invalid configuration — one with circular feature inclusions — would prevent the simple get-feature-config procedure given in Figure 5 from terminating. We will now consider the properties that must hold for a configuration to be valid and discuss how Wallaby verifies these properties.

5.2 Configuration validation: properties and algorithms

Recall that the Wallaby semantic model requires several properties of a valid configuration:

1. The graph implied by the feature-inclusion relation must be acyclic.
2. The graph implied by the parameter- and feature-dependence relations must be acyclic.
3. If feature F conflicts with F', both features cannot be installed on the same node. (Similarly, two conflicting parameters cannot appear in the configuration for a node.)
4. If feature F depends on F', a node whose configuration installs F must also install F'. (This also applies in the obvious way for parameters.)
5. Features cannot conflict with themselves. A feature cannot include or depend upon a feature that it conflicts with. (This applies for parameters as well, with the caveat that there is no inclusion relation for parameters.)
6. If the must-change flag is set in the metadata for parameter P, indicating that it must have a value supplied (rather than using its default value), any node setting P in its configuration must also supply it a value.

Ensuring these properties is conceptually straightforward but can be time-consuming on larger configurations. Group and feature definitions are intended to be modular, composable, and thus not necessarily complete; as a result, some validity invariants must be established for each node. We will first discuss basic procedures before presenting refined procedures that perform more efficiently and provide better user feedback in error cases. The algorithms we describe in this section depend on the auxiliary procedures described in Figure 6.

Name	Description
features-for-node(n)	returns a set of the features installed directly or transitively (viz., via feature inclusion) on every group that n is a member of
conflicts(f)	returns a set of features conflicting with f
dependencies(f)	returns a set of features that f depends upon

Figure 6. Auxiliary procedures called by configuration-validation algorithms

```
1: features ← features-for-node(node)
2: conflictHorizon ← ∅
3: for f in features do
4:     conflictHorizon ← conflictHorizon ∪ conflicts(f)
5: end
6: return features ∩ conflictHorizon = ∅
```

Figure 7. A procedure to determine whether or not the no-conflicting-features invariant is satisfied for a given node.

The first two properties are simple enough to establish. The underlying problem of cycle detection is trivial, and these properties need only be validated once for a configuration, rather than once for each node. For reasons that we shall discuss in Section 5.3, Wallaby ensures these properties hold as a side effect of generating topological orderings of the inclusion and dependency graphs for each entity.

Figure 7 presents the algorithm to ensure that a given node does not install two conflicting features. It does so by establishing a feature *conflict horizon*, or set of features conflicted with by features installed on a node, and then determining whether or not the node's feature conflict horizon has a non-empty intersection with the node's feature set. If this procedure is satisfied for every node, then it also trivially guarantees that any feature that is used in the configuration does not conflict with itself or include a feature that it conflicts with. In practice, the vast majority of features and parameters have few or no conflicts, so the performance of this procedure is not critical.

The procedure to ensure that nodes installing features with dependencies also install the features depended upon is in Figure 8. It is very similar to the procedure in Figure 7, with the exception that it accumulates dependencies across every installed feature for each node and determines whether or not the accumulated dependencies are a strict subset of the installed features. Note that if this procedure and the procedure in Figure 7 succeed for every node, that no installed feature transitively depends upon or includes a feature that it conflicts with.

Our final consideration is to ensure that parameters that must receive a user-specified value have done so. The procedure in Figure 9 returns the set of *node, parameter* pairs where the configuration for *node* fails to provide a required value for *parameter*. In the event that the configuration

```
features ← features-for-node(node)
depHorizon ← ∅
for f in features do
    depHorizon ← depHorizon ∪ dependencies(f)
end
return depHorizon ∩ features = depHorizon
```

Figure 8. A procedure to determine whether or not the installed-feature dependencies invariant is satisfied for a given node.

```
nodes ← all node objects
failures ← ∅
mc-params ← all parameter objects for which the "must-change" flag is true
for n in nodes do
    n-config ← get-node-config(n)
    for p in mc-params ∩ keys(n-config) do
        if n-config[p] = nil then
            failures ← failures ∪{(n, p)}
        end if
    end
end
return failures
```

Figure 9. A procedure to determine which nodes have failed to set parameters that require user-provided values.

succeeds at this validation step, this procedure will return the empty set.

5.3 Improvements and optimizations

The procedures we have described are straightforward (and thus fairly easy to understand and trust), but they are particularly inefficient. Furthermore, the configuration-generation procedures we presented don't provide a way to compose values from multiple features installed on a node, which can make defining certain kinds of list-valued or policy expressions difficult. In this section, we shall present some of the improvements, refinements, and optimizations that we have implemented in the Wallaby system.

More expressive apply

The simple apply procedure of Figure 4 handled redefinition of parameters by replacing preexisting definitions. Since apply was always used to introduce parameter bindings for a node in inverse priority order, this behavior ensures that the highest priority binding for a given parameter will be the one that survives to the final configuration and that all other bindings will be erased. This is perfectly acceptable for most cases, but in certain cases, we might wish to compose new parameter values with preexisting ones, if they exist. This is another consequence of the Wallaby philosophy that features should be modular and may be incomplete. We have identified two primary use cases for composing values:

1. *List-valued parameters.* Condor has a parameter called `DAEMON_LIST`, which is a comma-separated list of daemon processes that should run under the Condor master. Features that require new daemon processes should be able to add the processes that they require to `DAEMON_LIST` independently of changes made to `DAEMON_LIST` by other features.
2. *Policy expressions.* Some Condor parameters specify policy expressions, indicating that certain criteria must be met, for example, for a machine to become available for jobs, for a machine to choose to run a particular job, or for a machine to choose to evict and kill a job. These expressions often consist of conjunctions and disjunctions of simpler expressions; Wallaby features should be able to add a conjunct or disjunct to an expression defined in another feature.

The apply procedure that is actually in Wallaby allows users to define features that compose values when they are applied to preexisting configurations. Currently, the code treats several kinds of parameter values specially: those that begin with `>=` indicate that their value should be appended to an existing value as part of a comma-separated list, and those that begin with `&&=` or `||=` indicate that their value should be appended to an existing value as a clause in a conjunction or disjunction.

By way of example, consider two hypothetical features: *FooFeature* and *BarFeature*. *FooFeature* contains one parameter binding: `List = >= FOO`. Likewise, *BarFeature* also contains one parameter binding: `List = >= BAR`. A node with no other configuration whose identity group installed the features *FooFeature, BarFeature* would have the generated configuration `List = BAR, FOO`; since *BarFeature* is installed at the lowest priority, `>= BAR` is the first value that the parameter `List` gets. Because `List` is empty at first, `BAR` becomes its value (it is not appended to anything). When *FooFeature* is processed, applying the value `>= FOO` results in appending `FOO` to the preexisting value for `List`.

Optimizations and refinements

The Wallaby system implements several refinements to the basic algorithms presented above in order to provide better performance and, in some cases, more user-friendly output. In this section, we will present these improvements, roughly in the order that they were implemented in Wallaby itself.

The first improvement is to only calculate configurations for nodes whose configuration has changed since their last recorded configuration. Wallaby accomplishes this by maintaining *dirty lists* that record when an API call changes the state of a configurable entity: a node, group, feature, or parameter. At activation time, then, it only need consider nodes whose state may have changed as a result of API calls since the last activation. This can be a substantial savings when making simple, localized changes to large configurations.

The algorithms presented above calculate group configurations once for each node that is a member of that group. A better approach is to process each affected group once and generate a *partial configuration* that can be applied to a node's configuration and will have the same results, at any priority, as dynamically calculating the group's configuration and applying it to that node at that priority. In order to do so, we will need a procedure that calculates the configuration for a group; this is essentially similar to the inner loop of get-node-config from Figure 3, with the exception that it preserves parameter-value append markers like `>=`, so that a value that is to be composed in the group's configuration will also be composed (instead of replaced) when the partial configuration is applied.

Another effective improvement is to calculate each feature configuration before processing any groups and then caching these as applicable partial configurations. In fact, we can exploit the inclusion relationship between features in order to ensure that we calculate a partial configuration for any

given feature at most once. If we process features by visiting them in a reverse topological ordering of the graph implied by the inclusion relation, then we can guarantee that we will have a cached partial configuration for a feature F before we visit any feature F' such that F' includes F.

Many of the validity properties Wallaby enforces as invariants can fail to continue to hold as the result of a single change: introducing a cycle in the feature-inclusion graph, causing a feature to conflict with a feature that it includes or depends upon, etc. By checking these properties *proactively*, we can amortize the cost of checking these properties over every Wallaby API call that changes the state of the configuration store. Wallaby is able to identify certain changes that will always cause a valid configuration store to become invalid. This refinement not only improves performance at activation time, but it can also provide more useful error feedback by presenting Wallaby users with error messages immediately after a change that introduces inconsistency.

The final improvement we will discuss relates to how Wallaby stores versioned configurations for each node. These must be stored indefinitely so that clients can request the difference between old configurations, and should be available quickly. Approaches to dealing with generated data in network services typically fall on a spectrum between "baking," in which data is generated and cached whole, and "frying," in which data is generated on-demand. We have found that, in the case of complex Wallaby configurations, the best approach is what we have termed "parbaking" configurations: by baking partial configurations for each affected group at a given version and then storing a sequence of memberships for each node, we can quickly regenerate configurations on demand for a given node at a particular version. Due to the nontrivial overheads involved with serializing large node configurations, this can be a great performance improvement. We have observed this technique to perform twice as well as simply baking configurations for each node in tests of complex configurations on large pools; it will be the default in the next release of Wallaby.

6. The Wallaby API and tools

Wallaby is an efficient and expressive way to manage configurations for large Condor pools, but its real power subsists is in its programmability through an elegant networked API. In this section, we will discuss the API, the various configuration and monitoring tools that ship with Wallaby, and Albatross, a testing framework for Condor pools and policies that uses PyUnit and Wallaby.

The Wallaby API uses the Qpid Management Framework (QMF) as a transport layer, which presents an object-management interface and asynchronous event notification over AMQP. It publishes the following kinds of objects, some of which we discussed in greater detail in Sections 4.1–4.5:

1. *Store*, a single object corresponding to the main service, which supports methods related to creating and destroying other configuration entities; configuration validation, activation, and differencing; and snapshot loading and storing,
2. *Node*, one object for each node, modeling its name, group memberships, version number at last update, time of last check-in, and whether it was explicitly configured or just checked in and got the default configuration,
3. *Group*, one object for each group, modeling its name, installed features, and custom-set parameters,
4. *Feature*, one object for each feature, modeling its name; included, dependent, and conflicting features; and parameter settings,
5. *Parameter*, one object for each parameter, modeling its name, type, documentation, default value, and whether or not the user must supply an explicit value for this parameter in configurations that use it,
6. *Subsystem*, one object for each Condor subsystem, modeling the parameters that affect that subsystem,
7. *Snapshot*, one object for each user-created snapshot, modeling its name and creation date, and allowing loading of saved data into the store, and
8. *NodeUpdatedNotice*, an event object representing an asynchronous notification of configuration changes for a set of nodes.

The Wallaby QMF API is accessible in any language with QMF bindings; currently, this includes C++, Python, and Ruby, although the C++ implementation is designed to be accessible from other languages via the `swig` foreign-function interface. We provide idiomatic Python and Ruby client libraries that eliminate some of the complexity of dealing with remote objects and are simpler to use than the bare QMF interface. Finally, as we mentioned in Section 3, Wallaby includes an API client that publishes a RESTful HTTP service for reading configurations; a RESTful interface to all of Wallaby's functionality is under development. The API has been largely stable for over a year as of this writing and has proven flexible enough to support a wide range of applications: guided and batch configuration tools; pool management and analysis tools; automated node "tagging" with system information; and sophisticated, automatic functionality, policy, and scale testing. We will discuss some of these applications in the remainder of this section.

6.1 Guided Condor configuration tools

As part of the Wallaby project, we have built two types of configuration tools. Both of these use Wallaby's support for semantic configuration modeling, as well as a database we have compiled of Condor parameter metadata and high-level functionalities represented as Wallaby features, in order to ensure that generated configurations will be valid, sensible, and functional. The first kind we will discuss are interactive, guided tools with Condor-specific knowledge for simplifying common tasks.

The guided tools present a textual, menu-driven interactive interface. After each step, the guided tools present any configuration errors and warnings along with suggestions for how to resolve them, and prompt for the user's direction as to how to proceed. After any successful change, the tools offer the user an opportunity to save a snapshot of the store's state, in order to restore to it later.

The guided tools also include functionality to inspect features and parameters registered in the Wallaby service, to examine the live configurations of every node, and to list and inspect snapshots. Finally, these tools also include specialized shortcuts for configuring Condor functionality related to running virtual machines as jobs and running jobs on Amazon's Elastic Compute Cloud.

`activate`	Activates pending changes to the pool configuration.
`add-feature`	Adds a feature to the store.
`apropos`	Provides a list of parameters whose descriptions contain a keyword or match a regular expression.
`console`	Provides an interactive wallaby environment.
`dump`	Dumps a wallaby snapshot to a file.
`explain`	Outputs an annotated display of a node's current configuration.
`feature-import`	Imports a wallaby feature from a Condor configuration file.
`help`	Provides brief documentation for wallaby shell commands.
`http-server`	Provides a HTTP service gateway to wallaby node configurations.
`inventory`	Lists (a subset of) wallaby-managed nodes.
`list-snapshots`	Lists snapshots in the store.
`load`	Loads a wallaby snapshot from a file.
`load-snapshot`	Loads the snapshot with a given name.
`make-snapshot`	Makes a snapshot with a given name.
`modify-feature`	Alters metadata for a feature in the store.
`new-command`	Generates Ruby files containing templates for a new wallaby shell command.
`remove-feature`	Deletes a feature from the store.
`show-feature`	Displays the properties of a feature.

Table 1. Select Wallaby shell commands from the collection that are included in the Wallaby source distribution.

6.2 Batch-oriented configuration tools

Interactive configuration is not suitable for every application or installation, so Wallaby also provides batch-oriented tools that accomplish a single task and are suitable for scripting. These commands typically mirror API operations, exposing a single operation in a single command invocation. For example, there are commands to add a new feature with a certain set of properties, to modify a feature's properties, to show a feature's properties, and to delete a feature from the store. Similarly, there are analogous commands to create, inspect, delete, and update nodes and parameters. Each of these commands is a simple, command-line interface atop the Wallaby API, and each is implemented as part of the *Wallaby shell*, which we shall discuss next.

6.3 The Wallaby shell

The Wallaby shell is a solution to avoiding the boilerplate inherent in writing client applications for networked APIs. The Wallaby shell is an extensible command-line user interface to Wallaby, consisting of two main parts. First is the `wallaby` command, which handles common tasks like connecting to the AMQP broker, authenticating the user, and initializing the Wallaby client library before transferring control to a task-specific *subcommand*, which is implemented as a plug-in. The second part of the Wallaby shell is a library that makes it trivial to develop new subcommand plug-ins, simply by creating a Ruby class that satisfies a given interface. Wallaby itself includes many shell subcommands, a sampling of which are listed in Table 1. We will discuss some of these in greater detail, after providing an overview of how a developer might extend the Wallaby shell.

The interface that Wallaby shell command plug-ins must implement is quite minimal. In fact, to extend the Wallaby shell, a developer need only provide a Ruby class with the following four methods:

```
# FOO is set to BAR in feature ExecuteNode, which is
# included in feature OverprovisionedNode, which is
# installed on group HardWorkers, of which
# node1.example.com is a member.
FOO = BAR
```

Figure 10. Example `wallaby explain` annotation

1. A method that returns the name of the new shell command as a string;
2. A method that returns a one-line description of the new shell command;
3. `init_option_parser`, a method that returns an object to process command-line options; and
4. `act`, a method to do the actual work of the commands. By the time this method is invoked by the `wallaby` command, the client library is initialized and connected to the AMQP broker, with a reference to the Wallaby store client object accessible from an instance variable in the plug-in class.

The Wallaby shell interface also allows (but does not require) the developer to supply callbacks to run after subcommand initialization or command-line option processing. Wallaby shell command plug-ins can thus be very simple (on the order of a dozen lines of code to implement genuinely useful functionality), but are flexible enough to support sophisticated behavior. In fact, Wallaby includes a shell command to make it easier to define new commands: `wallaby new-command`, which generates a skeleton Ruby class that will work out of the box as a Wallaby command plug-in and is ready for a developer to fill in with actual command behavior.

6.4 Notable Wallaby shell commands

We now turn our attention briefly to a few interesting Wallaby shell command plug-ins; we have selected these to show the versatility and scope of the Wallaby API, as well as the simplicity and power of the extension mechanism afforded by the Wallaby shell.

We first consider `wallaby feature-import`, which turns a snippet of a Condor configuration file into a Wallaby feature definition and installs this feature in the Wallaby store, creating missing parameters as necessary. The snippet may contain specially-formatted comments to describe other metadata. Because Condor ignores these comments, it is possible to use the same file both as a Condor configuration file and as a Wallaby feature description. In our experience discussing the configuration needs of large Condor pools with their administrators, many Condor installations rely on customized systems for managing and deploying several Condor configuration file snippets to nodes depending on that node's requirements; thus, the `feature-import` command simplifies migration from such a legacy configuration tool to semantic configuration with Wallaby.

Configuration can be confusing, even in light of the semantic model Wallaby provides. The `wallaby explain` command simulates the configuration-calculation algorithm, producing a human- and machine-readable configuration file in which each key-value pair is annotated with a comment explaining how that value was calculated. An example explanation for a single parameter is given in Figure 10.

Because managed nodes check in with Wallaby at regular intervals even if they are not notified of configuration changes, its database of node metadata is useful for pool management

and inventory as well. The `wallaby inventory` command is designed to give a snapshot of the health of a Condor pool: it shows the name of each node; whether it was an explicitly-configured node or whether it checked in unconfigured and received the default group configuration; and when it last checked in. The `inventory` command can sort by most recent check-in (in order to identify failed nodes) or node name. It can also run with expressive constraints, such as:

```
memberships.size == 0 && name =~ /(foo|bar)[0-9]+/
```

which will list only nodes that have no group memberships and that have names containing the substring `foo` or `bar` followed by one or more digits, or

```
! last_checkin.is_never && last_checkin < 2.hours_ago
```

which will show nodes that have checked in at least once, but not in the last two hours. Finally, the `inventory` subcommand can produce machine-readable output, in order to more easily write scripts combining its output with that of Condor's comprehensive status-inspection tools.

The last command we'll examine is the simplest, but also the most flexible. The `wallaby console` command just provides an interactive Ruby environment with the Wallaby client library loaded and connected to the broker — that is, the very same environment that a new shell command would have inside its `act` method. This is useful for interactive experimentation with API methods, generating complex and repetitive features, and prototyping new Wallaby API clients. `wallaby console` can also be used as a UNIX script interpreter, so it is possible to make so-called "shebang" scripts that use the Wallaby console to handle connecting to the AMQP broker.

6.5 Automated pool testing with Albatross

Albatross[4] is an open-source unit testing framework for generating and testing Condor configurations. Albatross uses the Wallaby Python API to programmatically build Condor configurations and then apply them to Wallaby-managed nodes. By combining these Wallaby features with the Python PyUnit testing library, Albatross enables developers, testers, and administrators to run controlled, repeatable experiments involving Condor pool configurations.

We have used Albatross and Wallaby internally at Red Hat to systematically test Condor's correctness and performance with both default and customized configurations. We have also used Albatross to test the performance of the Cumin Condor management console[5]. We have evaluated both of these workloads successfully with a range of test pool sizes from fairly small — a single rack of modest servers — to fairly large — a large private clouds of thousand of multicore nodes. The Albatross framework enables the unit tests for these configurations to be quite compact. The tests are all less than 200 lines of code, and vary little with the scale of the configuration; frequently tests differ only in their parameters.

Albatross has proven especially beneficial for simplifying and automating the construction of large and repetitive features such as over-provisioned execute node configurations, in which a single node is configured with multiple Condor `startd` daemons in order to simulate a number of nodes that is larger than the number of physical machines available. (This capability is especially important for evaluating throughput and scalability limits of Condor itself and of the Cumin management console.) The number of Condor `startd` daemons, the number of execute slots per `startd` and whether those slots can be partitioned dynamically are all controlled by Albatross parameters. We used this facility to automatically generate valid multi-`startd` configurations of up to 1000 individual `startd` processes per physical machine, a task that would have been both labor intensive and error-prone to accomplish manually. It is similarly easy to generate multi-Scheduler configurations and multi-Collector configurations of any size, and apply them to chosen nodes.

Albatross also uses the Wallaby API to automatically save both pre-test and testing configurations, whose names are logged to allow future reconstruction of testing configurations. Pre-test configurations are automatically restored after a test is run. Saved testing configurations can also be optionally loaded by Albatross to run a unit test using a pre-defined configuration. These features would be difficult or impossible to support without the configuration versioning provided by Wallaby.

Finally, Albatross is not merely a tool for Condor developers who wish to perform directed scale experiments and correctness tests. It is also useful for administrators who wish to evaluate and compare different scheduling policies. We expect that, in the future, we will also use Albatross for automated functional and regression testing suites, including tests with random but valid configurations in order to explore uncharted parts of the Condor test and configuration space.

7. Experience and evaluation

Wallaby is a mature open-source project that is also available with commercial support as part of Red Hat Enterprise MRG product. The current release of the Wallaby open-source project[6] consists of approximately 9,000 lines of Ruby, not including support libraries like the custom object-management and persistence libraries. It includes a comprehensive specification and test suite with over 500 executable examples of correct application-level behavior and acceptable performance under stress; these specifications and tests are derived both from the application design and from defect reports filed against earlier versions of Wallaby. This test suite has given us the freedom to quickly evaluate the performance and correctness of even fairly drastic implementation changes; for example, all of the enhancements discussed in Section 5.3 were validated with the aid of the test suite.

Since the earliest stages of developing Wallaby, we have used it internally to manage our development and testing Condor pool. In fact, we started doing so even before Wallaby had some useful functionality, like configuration differencing. As a result, we were able to find scale and correctness problems and clarify and improve the API even before much outside adoption of the open-source Wallaby project or the Red Hat-supported version. Unsurprisingly, we found this approach was successful: having ourselves and our coworkers as internal customers drove the quality and expressiveness of the project. We also received regular, valuable, and encouraging feedback from team members who were new to Condor configuration and yet found it easy to use Wallaby to configure the pool for special projects or tests. In general, the open nature of the project has been a huge boon: we have been able to incorporate suggestions and implement feature requests originating from internal users, Red Hat customers, and members of the open-source and Condor communities.

We have evaluated the performance of the Wallaby store under stressful conditions in three ways. First, via Albatross

[4] `http://git.fedorahosted.org/git/grid/albatross.git`

[5] `https://fedorahosted.org/grid/wiki/Cumin`

[6] Version 0.10.5, as of this writing

experiments, which used Wallaby to configure test pools ranging from a small pool of 64 8-core nodes managed by 1 scheduler up to a moderately large pool of 5000 8-core nodes managed by 15 schedulers. Wallaby met these scale demands and was able to support Albatross at these reasonable scales. Because these experiments can overprovision single physical machines in a way that most real-world installations would not, they are an especially good test of unusually large per-node configurations. Our second performance evaluation technique is part of the Wallaby test suite, which includes a specially-generated stress test that must complete with acceptable latency to succeed — that is, it must run quickly enough that an API client would not time out. This test repeatedly reconfigures and activates 2000 nodes that are each provisioned with 90 `startd` processes; to configure 90 `startd`s alone requires over 460 parameters. (One author regularly runs this test successfully on a modest portable computer.) Finally, we have developed a mechanism for generating stress tests based on randomly-generated valid Wallaby configurations that satisfy certain probability distribution characteristics: we can specify, for example, the mean, variance, and distribution for the number of features a typical feature will extend or conflict with, the number of groups a node is a member of, the number of features a group will have installed, and so on.

8. Related work

This paper has presented Wallaby, a special-purpose semantic configuration system enabling configuration of Condor pools. While there are many open-source and proprietary systems for managing distributed application configurations without semantic validation and several research systems for defining constraints that must hold over application configurations, Wallaby is the only system we know of to combine the following desirable features:

1. It supports *semantic* configuration management and validation;
2. It features a rich programmable API that has demonstrated its usefulness for applications beyond configuration management (including pool inventory and experiment design);
3. It is scalable along two dimensions: we have demonstrated its suitability for fairly large numbers of managed nodes running particularly large and complex generated configurations; and
4. It is a robust system that is used in real-world applications and is readily available as an open-source project.

We will now briefly examine some notable related efforts.

The Rocks Cluster project [7] provides a software distribution, provisioning system, and configuration system for compute clusters. This project targets a rather broader scope (installation, provisioning, and multiple components) than Wallaby, but its configuration model is rather more limited: configuration parameters are defined in "rolls" which are pushed out to machines as part of their provisioning. When a machine is to be reconfigured, it is reprovisioned, which could prove expensive in utility computing environments. Rocks and Wallaby are complementary efforts, however — certainly a Rocks system could use Wallaby for dynamic, semantic Condor reconfiguration.

Widely-used open-source tools like Puppet, Chef, and `cfengine` provide distributed configuration deployment capabilities and some (typically imperative) mechanism for synthesizing configurations from smaller snippets. However, these systems do not support the sort of semantic modeling possible in Wallaby, meaning that per-subsystem reactivation, a priori configuration validation, and other useful features are unavailable. Like Rocks, though, these systems could be used to manage other parts of a distributed system in cooperation with Wallaby.

The PRESTO system of Enck et al. manages configurations for network devices and services. PRESTO includes a templating language that enables users to develop modular pieces of configuration, called *configlets*. This language is similar to dynamic-document template languages that allow, for example, web pages to be generated from multiple container templates, except it generates configuration files instead of web pages and includes support for type-checking values like IP addresses, netmasks, hostnames, and ranges of numbers. The PRESTO system uses a two-phase approach to identify potential errors in specified configurations and request corrected meta-parameters from system administrators. Unlike Wallaby, it can manage several different system components (not merely Condor), but its model is, perhaps necessarily, more complex and less declarative.

The PoDIM system [2] includes a configuration constraint language and support for modeling configuration entities in the Eiffel programming language. It is general-purpose and can collaborate with an existing distributed configuration deployment tool like `cfengine` or Puppet. The SmartFrog system [4] manages distributed software systems of several components; it also uses an object-oriented programming language to specify configurations. The main difference between these systems and Wallaby is that Wallaby provides a simpler, declarative model for specifying configuration entity constraints; Wallaby is able to do this easily because it is a special-purpose system.

References

[1] *2001 IEEE International Conference on Cluster Computing (CLUSTER 2001), 8-11 October 2001, Newport Beach, CA, USA*, 2001. IEEE Computer Society.

[2] T. Delaet and W. Joosen. Podim: a language for high-level configuration management. In *Proceedings of the 21st conference on Large Installation System Administration Conference*, pages 21:1–21:13, Berkeley, CA, USA, 2007. USENIX Association.

[3] W. Enck, T. Moyer, P. Mcdaniel, S. Sen, P. Sebos, S. Spoerel, A. G. Greenberg, Y. wei Eric Sung, S. G. Rao, and W. Aiello. Configuration management at massive scale: system design and experience. *IEEE Journal on Selected Areas in Communications*, 27:323–335, 2009.

[4] P. Goldsack, J. Guijarro, S. Loughran, A. Coles, A. Farrell, A. Lain, P. Murray, and P. Toft. The smartfrog configuration management framework. *SIGOPS Operating Systems Review*, 43:16–25, January 2009. ISSN 0163-5980.

[5] M. Litzkow, M. Livny, and M. Mutka. Condor - a hunter of idle workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, June 1988.

[6] U. of Wisconsin Condor Team. *Condor Version 7.6.0 Manual*, 2011.

[7] P. M. Papadopoulos, M. J. Katz, and G. Bruno. Npaci rocks: Tools and techniques for easily deploying manageable linux clusters. In *CLUSTER* DBL [1], pages 258–.

[8] S. Vinoski. Advanced message queuing protocol. *IEEE Internet Computing*, 10:87–89, November 2006.