ELASTIC BUILD SERVICE

Ville Seppänen
Tieto Corporation / Tampere University of Technology
P.O. Box 527, 33101 Tampere, Finland
ville.t.seppanen@iki.fi

Abstract

Linux-based operating systems such as MeeGo consist of thousands of modular software packages. Compiling source code and packaging software is an automated but computationally heavy task. As the load on a build cluster can vary greatly, a local infrastructure is difficult to provision efficiently. In this paper the elastic acquisition of cloud resources is presented as a means to ensure sufficient computing capacity for a software build system in case of rapid growth in the popularity of MeeGo. A proof-of-concept implementation was done using a specific build system commonly used for MeeGo development. This system is Open Build Service, a centrally managed distributed build system capable of building packages for several distributions and architectures. Main concerns were the technical feasibility, security and efficiency of the proposed solution. A script was implemented to autonomously manage the elastic cloudbursting, monitoring resource usage and demand and making decisions whether new machines should be requested or idle ones terminated. The implementation was evaluated by comparing build times with and without the support from the cloud. The latencies incurred by the physical distance to the cloud were not insurmountable and the system scaled up in a matter of minutes. The main advantage achieved with cloud usage in this work was the advent of seemingly infinite number of resources on-demand, ideal for handling sudden bursts of packages that can be built in parallel.

Keywords: Cloud computing, cloudbursting, build system, software build, OBS, Open Build Service, computing cluster, distributed system, virtualization

1 Introduction

Operating systems have grown in complexity over the last ten years. Modern Linux-based operating systems consist of thousands of modular software packages. These packages are created with automated build tools which require a lot of computational power. Furthermore, the building needs of software developers may vary greatly, causing irregular load spikes on the build system. This makes it difficult to successfully provision a dedicated infrastructure; overprovisioning the build cluster to cope with even the largest temporary spikes would be costly, while underprovisioning would lengthen build times during load spikes.

Cloud services, Infrastructure-as-a-Service more specifically, provide easily obtainable, utility-like computing power over the Internet in the form of customizable virtual machines. Users can request resources from the cloud as necessary, usually over an Application Programming Interface (API) or a web-based user interface, and pay based on the actual usage.

As the growth of popularity with MeeGo (or Tizen) is uncertain, means to ensure software build capacity needs to be solved proactively. In this paper we present the elastic acquisition of cloud resources as a means to ensure sufficient computing capacity for a software build
system. This system is Open Build Service (OBS), a centrally managed distributed build system capable of building packages for MeeGo among other distributions. OBS is used extensively in MeeGo development, but current build clusters are on static, dedicated hardware. Successful cloud usage would smooth load peaks and buy time for additional hardware investments. Main concerns were the technical feasibility, security and efficiency of the proposed solution.

2 Cloudbursting – Elastic hybrid clouds

The key motivations for cloud customers are the economic benefits from achieved elasticity and transference of risk in regards of provisioning. Cloud computing can be seen as transforming capital expenditures into operational expenditures. Maintaining an infrastructure that can sustain even the highest peaks can be costly, especially if the peaks are rare and far larger than the average load [1]. Constant cloud usage may also cost more in the end, but the customer is free from the risks of under- and overprovisioning hardware [2, p. 10].

Multiple public or private clouds can be used in conjunction, forming a hybrid cloud. The underlying use of several clouds is often hidden from the end-user, but provisioning should take into account the heterogeneity of the providers: their service level agreement, billing and current state. Because of this, hybrid clouds can be more difficult to manage effectively.

Cloudbursting is a concept of expanding a pool of local resources into a public cloud when local capacity reaches its limit. It essentially creates a scalable and elastic hybrid cloud infrastructure, on-demand without manual intervention [1]. This is especially efficient in handling load bursts; allowing extra jobs to overflow to the cloud and making sure applications remain available when local resources become saturated. Marshall et. al [3] have presented a model of an elastic site for describing cloudbursting, shown in Figure 1.

![Figure 1. Elastic site model](based on 3)
In this model, the resource usage of a static, local site is monitored by a manager component and additional resources are requested from the cloud based on the demand. This forms an elastic site, capable of adapting to fluctuating resource demand. A similar resource manager is implemented in this work, termed as a service manager in accordance with RESERVOIR’s reference architecture of cloud management [4]. In this architecture, management software is divided into three layers: Virtualization Hypervisors that manage virtual machines on a hardware server; Virtual Infrastructure Managers (VIM) that control large pools of virtual machines even spanning over multiple separate clouds; and finally Service Managers that monitor the needs of overlying services and adjust the amount of resources accordingly.

Just as ISPs use multiple network providers to backup their service, very high availability can be achieved only by using multiple cloud providers [2, p. 14]. VIMs are needed when allocating virtual machines elastically on multiple clouds.

3 Software building with Open Build Service

Software building is a process where software is compiled from source code and bundled with configuration data into packages. Packages are very prominent in Linux-based operating systems, where applications and the operating system itself are composed of packages. Packages are used to ease the installation, updating and uninstalling of software. In addition to the actual code, a package holds metadata such as package name, version, how to install and uninstall it and what dependencies it has. To lessen the amount of redundant code, packages can depend on other packages, forming a dependency tree. This metadata originates from the recipe (In RPM, this file is called a spec file) of the package, a file that holds the metadata and instructions on how to build the package.

To decrease build times, packages can be built in parallel. If one of the prerequisite packages is being built, the depending build job is blocked until its build environment can fulfill all dependencies. The dependency tree varies for each package, but there are often bottlenecks, large packages that many other packages depend on, which can block a great number of jobs until it is finished. The number of jobs that can be built at a time varies greatly, therefore justifying the need for elastic capacity. Depending on the hardware, build environment and what is actually being built, a build job can take from a minute for a single small package to hours or days for building a complete operating system.

Open Build Service (OBS), is an open-source, cross-distribution development platform [5], which is used to build software packages for several Linux distributions (e.g. MeeGo and openSUSE) and for different architectures (e.g. ARMv7 and x86). Software developers write source code and package descriptions and the system automatically goes through the dependencies and rebuilds necessary packages. Due to the modularity of packages, they can be built in parallel. OBS is a distributed system, where a head node manages the system and dispatches build jobs to multiple workers. The head node composes from several software services, which can be spread to separate hardware [6], a moderately scaled out example is shown in Figure 2. In smaller setups (such as in this work), the management components and data storage can be put on a single server.
The front-end consists of a web server and an XML API service, which is used by the command line client (OSC). The storage node has source code repositories and a database for persistent data. When developers make changes to the source code, the source code service notifies the back-end. The back-end hosts most of the decision-making components of OBS. When the job scheduler gets notified of source code changes, it calculates build dependencies and generates jobs into a queue for the dispatcher. The dispatcher assigns available jobs to idle workers. When a job is completed, the resulting package is optionally signed as authentic and then published to a repository from where the end-users can download the packages.

A worker is a Linux process, running on a network host called a build host. Each worker can only build a single package at a time, and by default the number of workers on a build host equals to the number of its processor cores. When a worker is assigned a build job, it downloads prerequisite packages from the repository and sets up a build environment. This has to be recreated for each build to ensure reproducibility. The worker downloads source files and compiles and packages the code. The resulting package is then sent to the package repository. Depending on the hardware, a build job can take from a minute for a single small package to hours or days for building a complete operating system.

4 Extending MeeGo building to the cloud

The overall requirement for the build system was that it would be locally hosted and would smartly use cloud resources for extra capacity when needed. As workers also exist in-house, auto-scaling features of IaaS providers cannot be used. Furthermore, a single worker can only build a single package at a time, no matter its instantaneous resource usage. A worker that is
building a package is completely occupied, even though its resource utilization would be temporarily low. Demand for new resources cannot be therefore determined from the usage of resources at that time. This requires monitoring of both the work queue and the build jobs, and allocating resources accordingly.

Based on the elastic site model and the RESERVOIR’s cloud management reference architecture, several functional components were picked to form a proposed architecture for an elastic build service, presented in Figure 3. The white boxes represent functional modules with their example implementations in parentheses.

Due to time constraints, the implemented architecture was simplified. Support for multiple parallel IaaS providers was dropped, affecting fault tolerance. Build hosts were also divided into two groups: Local Build Hosts (LBH) were created on static hardware and managed manually, while Cloud Build Hosts (CBH) were launched only in the Elastic Compute Cloud (EC2) of Amazon Web Services (AWS). Monitoring and management of CBHs were also done using AWS with a custom service manager. With these simplifications, a separate VIM is not needed.

With small and simple OBS deployments, the various management server functions introduced earlier (front-end, back-end and storage in Figure 2) can be deployed as a single server. The implemented system is shown in Figure 4.
The local infrastructure was set up on virtual servers on two commodity-class desktops (quad core with three cores dedicated to the virtual server): one running all the OBS management functions, referred to as OBS server and the other as an LBH running three worker processes. The service was created from the latest stable OBS appliance image loosely following the OBS and MeeGo community-made instructions. Once the build service was working locally, the cloud extension was added.

Each virtual machine in EC2 is created from an Amazon Machine Image (AMI). The CBHs would be launched from a generic CBH AMI and then customized during boot-time using the User Data mechanism of EC2. To do this, a virtual machine was launched from a public openSUSE AMI, configured to run OBS worker and connect to the OBS server through a VPN tunnel, and bundled back into a new AMI. Virtual machines manually launched from this custom AMI would boot up, start the worker program(s), connect to the OBS server and start building packages.

A simple controller for automating the elasticity was then needed. OBS itself handles scheduling and dispatching centrally, but is not capable of spawning new workers, let alone new virtual machines hosting the workers. The autoscaling feature would need to communicate tightly with the OBS server, thus a simple service manager script was needed. It was implemented in Python as both AWS and OBS can be interface through feature-rich Python modules, open-source boto [7] and OSC respectively. Connections between these software modules are shown in Figure 5.
The service manager was designed so that it could be run manually or scheduled with cron job scheduler. When it is executed, it gathers all the necessary data it needs to understand the current situation and makes decisions on whether it should launch or terminate CBHs or do nothing. The manager script has been made publicly available under GPL [8].

First, the service manager connects to AWS management and monitoring servers and requests metadata of existing CBHs. This data includes virtual machine ID, name tag, state, launch timestamp, IP address and the latest measured CPU utilization. Second, the service manager connects to the local OBS server and requests metadata of both the current workers (hostname, state, packages being built if any) as well as the queue of future jobs (number of jobs waiting and number of blocked jobs).

As a virtual machine can only be referenced in EC2 by their virtual machine ID (generated by EC2 itself), and as workers running on the virtual machines are only identified in OBS by their hostname, these two datasets need to be mapped to each other. OBS only manages workers and cannot start or shutdown build hosts, while EC2 can only manage virtual machines (i.e. the CBHs) but is not aware of workers.

When requesting a new CBH, the service manager generates a unique name using the UNIX timestamp (e.g. cbh-08808583). This name is then attached to the virtual machine as an EC2 tag, a key-value pair which allows arbitrary metadata to be stored. In addition, a short shell script is passed to the machine via EC2 User Data which changes the hostname of the machine to be the same as in the tag. This way, a CBH that is unnecessary according to the OBS API can be terminated in EC2 with its virtual machine ID.

After collecting data of the current situation, the service manager goes through all the found CBHs to be nominated for termination. Build hosts that are not connected to the OBS server or that are idle even though there is work to be done will be nominated. However, as it takes a varying amount of time for an instance to start up, create a VPN connection and connect to the OBS server (usually 3 minutes), newborn build hosts will be ignored until they reach a boot time threshold (set to 5 minutes).
Idle hosts that have no more work are also terminated if they are close to the end of a billing period. Virtual machines in EC2 are each billed hourly, for every full hour counting from the time the machine was started. Starting a new machine and letting it run for ten minutes and repeating it five more times in one hour, will cost six instance-hours instead of one. Starting and stopping instances should therefore be minimized. Idle build hosts will be kept waiting for new jobs until their lifetime approaches another full hour.

When the possible non-working build hosts have been terminated, the service manager determines whether it should request new build hosts. Each machine that is considered still booting up is reserved to have a job from the queue. Subtracting the number of booting workers from the number of jobs in the queue equals to the number of how many more workers are needed. The manager also has limits for the total number of build hosts and for the total number of hosts that are starting up. Most of the limits and threshold variables in the script can be easily fine-tuned to balance between performance and cost. Finally once the worker pool has been updated, the script writes any changes made to a log file.

5 Evaluation

One of the concerns in this research was that preparations for building in the cloud would take so long that it would not be feasible to outsource builds. Several measurements were performed to evaluate the speed gain and the cost when cloudbursting. A set of 30 randomly picked MeeGo packages were built first with only LBH and then with both LBH and CBH. Each scenario was repeated three times for averaging. On average, it took the LBH 153 minutes to build the batch, while with CBHs it took between 55 minutes (medium-sized high-CPU EC2 machines “c1.medium”) and 61 minutes (small EC2 machines “m1.small”).

The system uses an aggressive elasticity policy i.e. it gives explicitly a new worker for each job until the max number of workers is reached. This policy is used because of its simple implementation, and while it costs extra, it does minimize the overall build time. However, when changes in the source code occur, it often reflects into several packages through build-time dependencies. These packages have to be built in order, causing some jobs to wait for others to finish. Because of this, all packages cannot be always built in parallel.

Moreno-Vozmediano et al. have studied elastic cluster management both with an Embarrassingly Distributed computing cluster benchmark as well as a web server cluster [9]. In both scenarios, server jobs are highly parallelizable. With OBS however, the problem is that build jobs are large and thus last a long time, and jobs are often dependant on each other. OBS has a feature to build the large bottleneck packages on specific, high-powered build hosts. A successful build system should balance between the performance and number of workers.

In total, 60 packages were built in the cloud in these tests. This cost $3.14 for disk and snapshot usage, $2.66 for network and $17.10 for computing itself. The traffic is a small portion of the total (with latest prices even lower) even though this setup did not cache any packages in the cloud. A cloud-side cache could be setup, shared between CBHs, to slightly lower the traffic cost and time spent.

In general, build hosts can be distributed over the Internet without code changes and with very little configuration in OBS. With a virtual network, the build hosts are easily connected to the OBS server. Smarter elasticity policies (e.g. do not launch a new instance if some job is about to finish) should be implemented, but they require extra data from OBS, including estimates
of build durations. For cloudbursting purposes, worker status metrics from OBS are quite coarse. It does not provide any time estimates for the builds and is also slow to remove build hosts that have been terminated from the system. The free monitoring service in AWS is also too coarse, with its 5-minute refresh intervals. The main concern with AWS was however its inconsistent behavior. Sometimes instances failed to start up properly at a specific availability zone, not connecting to OBS server, not responding to connection attempts and not showing any activity in their system log. These machines were stated “running” and thus still billed by AWS.

Separate monitoring software is needed in conjunction with Virtual Infrastructure Manager (VIM) such as OpenStack to ensure service availability and to avoid provider lock-in. VIMs separate the OBS completely from the details specific to IaaS providers, allowing easy expansion to several IaaS clouds.

6 Conclusions

This paper presented how the infrastructure of a software build system can be distributed into domains that are geographically apart and owned by different organizations. With cloudbursting, this infrastructure dynamically expands as needed, lowering the costs of cloud usage by maximizing the use of acquired resources. Even though cloud services do provide cost-efficient flexibility, they have their challenges in confidentiality, integrity and availability. Whether to keep a system in-house or move it to a cloud depends on many things, and it should be assessed whether cloud usage fits the use case. A hybrid model is a convenient way to keep business-critical data in-house, while buying extra computational power from the cloud.

Another aspect to consider is the cost, and it is not simple to estimate. With public infrastructure services, the share of traffic and storage is minimal compared to the cost of computation needed for the actual building. For constant needs, large enterprises that already have the means and knowledge are better off setting up a build cluster of their own. Cloud providers make profit by utilizing economies of scale, amassing servers together to save up in infrastructure and management costs.

Open Build Service is a powerful and flexible tool for building packages for multiple device architectures and target distributions. The main problem in its cloud usage is the dependencies between packages and the great variance in build job times. The balance between the number and performance of workers needs to be optimized. Another issue is that Open Build Service has been designed for closed, trusted environments. It uses virtualization to sandbox the build environments on build hosts. This is to protect the build system from attacks originating from malicious open source code being built on the system, injected into the builds through package dependencies. Full-blown virtualization cannot be used in a cloud where build hosts themselves are already virtual machines, and support for lighter, OS-level virtualization such as Linux Containers (LXC) is needed. Without adequate protection, these attacks could deny the service or alter the contents of future packages built in the service.

The cloudbursting build system setup in this work was a simplified proof-of-concept. With the implemented service manager, the system automatically accommodates to the number of waiting jobs, spawning new build hosts when needed and terminating them when not. While the machines from the cloud are not superior in performance, the fact that there is an seemingly endless pool of them makes the system good in flushing out a long queue of jobs.
that can be built in parallel. For large bottleneck packages, separate powerful build hosts need to be set up as packages can depend on each other, therefore blocking other build jobs temporarily.

For production-use, however, the system requires more work in terms of security, availability, manageability and performance. Infrastructure and service management software are needed in multi-tenant, multi-service and multi-provider environments. There are several competing alternatives for Virtual Infrastructure Management, but service managers seem to lack options. As services vary greatly, it is difficult to make a generic API for managed services. Currently most services seem to rely on ad hoc service manager implementations.

All in all, cloud services have a definite support role in build systems and other parallel batch job processing systems. Running software services on cloud-compatible platforms make it possible to freely choose the amount of cloud usage with little changes. This kind of decoupling of software services from the underlying operating systems of separate computers can be perceived in modern cloud technologies.

References

5. Open Build Service (OBS). http://openbuildservice.org/