INRIA

CONCOURS EXTERNE D'INGENIEUR DE RECHERCHE

DIS-4

BAP E – INFORMATIQUE ET CALCUL SCIENTIFIQUE

Ouvert au titre de 2009
Arrêtés du 15 avril 2009 parus au JO du 23 avril 2009
Epreuve écrite
Note sur 20 – Coefficient 3 – Durée trois heures

le 22 juin 2009 de 14h à 17h

Le candidat peut traiter les exercices dans l'ordre de son choix.

Veillez à respecter l'anonymat dans les réponses.

Une attention particulière sera portée à la qualité de la rédaction, de la présentation, à l'orthographe et à la grammaire.

Exercice 1 : (1 point)

Voici 2 versions de la fonction factorielle.

```
ent factorielle(n) {
        assert (n>0);
        si n=1 alors
                retourner 1;
        sinon
                retourner n* factorielle (n-1)
        fin
}
```

Algorithme 1

```
ent factorielle (n) {
        assert (n>0) ;
        var ent acc = 1
        var ent i = n
        tant que i != 1
                acc = acc * i ;
                i = i -1 ;
        fait
        retourner acc ;
        }
```

Algorithme 2

Q1.1: Pour chaque algorithme, donnez le nom du style de programmation mis en œuvre.

Q1.2 : Quelles différences y a-t-il à l'exécution entre ces deux versions ?

Exercice 2 : (4 points)

Il est rare de devoir écrire un logiciel à partir de rien. En général, un logiciel est le résultat de la composition de plusieurs composants logiciels. Ceux-ci peuvent se trouver sous différentes formes, dont :

- les programmes exécutables, sous forme binaire
- les bibliothèques statiques ou dynamiques (vision système)
- les systèmes de plugins (vision logiciel)
- les services distants

Caractérisez brièvement chacune de ces formes quant aux conséquences sur la licence de votre logiciel, et l'impact à l'installation comme à l'exécution.

Exercice 3 : (6 points)

Attention : pour cet exercice, vous devez utiliser l'annexe A

Un projet européen vise à l'intégration de logiciels développés par différents partenaires. Un des partenaires est éditeur de distribution Linux, il s'est engagé en cas du succès à intégrer le résultat dans sa distribution. Toujours en cas de succès, les industriels du projet envisagent de créer un consortium pour partager entre eux les frais de maintenance du résultat du projet. Lors de la négociation du projet, la commission européenne suggère fortement de travailler avec un outil d'intégration continue. Le responsable de projet n'y connait rien et vous demande de lui rédiger une note d'une page ***en français*** (750 mots maximum) et un résumé de 10 à 15 lignes ***en anglais*** (200 mots maximum) précisant les objectifs d'un tel processus, les enjeux dans le choix d'un outil parmi ceux décrits dans les documents de l'annexe A ainsi que vos recommandations.

Exercice 4 : (2 points)

Votre responsable vous demande de faire un tableau de bord montrant les progrès dans le déroulement d'un jeu de tests. Vous disposez d'une fonction permettant de demander à un serveur central la liste des résultats de tests qu'il a reçu, avec pour chaque test le système d'exploitation sur lequel il s'est déroulé et son résultat. Vous connaissez par ailleurs le nombre **total** de tests à effectuer, qui est le même sur chaque système d'exploitation.

Le format de la liste des résultats est une série de lignes au format

"OS","nom du test","résultat"

où résultat est soit OK ou NOK

Le tableau de bord présentera les résultats de la manière suivante :

Nombre de tests réalisés : 135
- Nombre de tests OK : 132
- Nombre de tests NOK : 3

Nombre de tests restants 165

Windows XP:
- Nombre de tests réalisés : 35
  - Nombre de tests OK : 32
  - Nombre de tests NOK : 3
- Nombre de tests restants 65

Mac OS X Tiger:
- Nombre de tests réalisés : 50
  - Nombre de tests OK : 50
  - Nombre de tests NOK : 0
- Nombre de tests restants 50

Linux Ubuntu 09.04:
- Nombre de tests réalisés : 50
  - Nombre de tests OK : 50
  - Nombre de tests NOK : 0
- Nombre de tests restants 50

La mise en œuvre suivante **(***Algorithme 3***)** est esquissée au tableau blanc lors de la conversation.

Commentez les limites de la solution adoptée (sur le principe de la solution proposée, et non sur les détails de sa mise en œuvre) et proposez une meilleure approche en décrivant votre solution dans un langage de programmation fictif proche de celui de l'algorithme 3.

```
NB_TOTAL_TEST = 100 ; NB_OS = 3;
/** Obtenir le nombre total de tests effectués */
Handle = server.getlogs() ;
nb_tests = nb_ok = nb_nok= 0 ;
while (line = handle->getLine()) {
        nb_tests++ ;
        if (line->getResult() = "OK") {
                nb_ok++;
        }else {
                nb_nok++ ;
        }
}
Print "<TAB>Nombre de tests réalisés : " + nb_tests
        Print "<TAB>Nombre de tests OK :" + nb_ok ;
Print "<TAB>Nombre de tests NOK :" + nb_nok ;
Print "Nombre de tests restants: "; Print NB_OS*NB_TOTAL_TEST – nb_tests ;
/** Obtenir les statistiques de chaque système d'exploitation */
foreach os in "Windows XP", "Linux Ubuntu 09.04", "Mac OS X Tiger" {
        print os ;
        Handle = server.getlogs() ;
        nb_tests = nb_ok = nb_nok= 0 ;
        while (line = handle.getLine()) {
                if (line.getOS() = os) {
                        nb_tests++ ;
                        if (line.getResult() == "OK") {
                                nb_ok++;
                        } else {
                                nb_nok++ ;
                        }
        }
        Print "<TAB>Nombre de tests réalisés : " + nb_tests
                Print "<TAB><TAB>Nombre de tests OK :" + nb_ok ;
        Print "<TAB><TAB>Nombre de tests NOK :" + nb_nok ;
        Print "<TAB>Nombre de tests restants: "; Print NB_TOTAL_TEST – nb_tests ;
}
```

Algorithme 3 : Script de synthèse des progrès d'un serveur de test

Exercice 5 : (1 point)

Donnez un programme, dans le langage *fonctionnel* de votre choix, équivalent à la méthode Java suivante. Votre implémentation doit être *récursive terminale (tail-recursive)* :

```
static int f(int x, int y) {
   int aux=x;
   int res=0;
   while(aux!=y){
      res = res + aux;
      if (aux<y) aux=aux+1;
      else aux=aux-1;
   }
   return res;
}
```

Exercice 6 : (2 points)

Un logiciel de calcul symbolique doit manipuler des expressions de la *logique des prédicats* contenant les connecteurs : ∀ (pour tout), ∃ (il existe), ∧ (et), ∨ (ou), → (implique), ¬ (non). Notez que ∀ et ∃ sont des *lieurs*, et qu'il est possible de lier plusieurs fois le même nom de variable dans la même expression.

Q6.1 Donnez deux exemples de formules logiques dans lesquelles le même nom de variable est lié deux fois.

Q6.2 Proposez une structure de donnée, dans le langage fonctionnel de votre choix, permettant de manipuler de telles expressions. Vous expliquerez (5 lignes maximum) comment votre structure permet de gérer les lieurs et le fait que plusieurs variables peuvent avoir le même nom.

Q6.3 Donnez le code de la fonction *rename* prenant une expression en argument et retournant une expression équivalente ne contenant pas de nom lié deux fois.

Exercice 7 : (4 points)

Soit un petit langage de programmation contenant les expressions et instructions suivantes (l'évaluation de ces instructions est standard) :

Expressions arithmétiques:

E ::= x | n | E + E | E * E | E – E

où **n** est un entier quelconque et **x** un nom de variable quelconque

Expressions booléennes:

B ::= x | true | false | B or B | B and B | not B | E < E

Instructions:

I ::= x:=E | x:=B | If B then I else I | while B do I done | I ; I | begin I end

On désire implanter *un algorithme de typage statique et fort* pour ce langage. Comme il n'y a pas d'instruction de *déclaration* de variable, le type de ces dernières doit être *inféré* par l'algorithme en analysant les instructions selon les principes suivants:

1. Une variable ne peut pas être utilisée avant d'avoir été initialisée par une affectation.
2. L'affectation d'une variable détermine son type, toutes les affectations d'une variable doivent correspondre au même type.

Les deux types possibles sont *int* et *bool*.

Exemples:

```
x:=2;
if x < 3
then y:=5
else y:=3;
x := y + 3;
```

est bien typé (x est de type int et y est de type int).

```
x:=2;
if x < 3
then y:=5
else y:=true;
```

est mal typé car y n'a pas le même type dans le then et le else.

Q7.1 Décrivez, en 10 lignes maximum, les principes des langages à typage fort (*strong typing*). Vous présenterez les avantages et inconvénients, selon vous, de ces langages par rapport aux langages à typage faible. Vous donnerez également des exemples de langages appartenant à ces deux catégories.

Q7.2 Proposez un système typage fort (sous forme de règles d'inférence) des instructions selon les principes énoncés plus haut. Le système de typage devra déterminer si un programme est bien typé ou non. Vous donnerez une règle de typage par expression et par instruction. Les jugements de typage concernant les instructions seront de la forme:

$$\Gamma \vdash I:\Gamma'$$

Les jugements de typage concernant les expressions seront de la forme

$$\Gamma \vdash e:T$$

où Γ est l'environnement de typage dans lequel l'expression e doit être typée et T son type.

Annexe A

Extrait de http://en.wikipedia.org/wiki/Continuous_Integration

Continuous integration

From Wikipedia, the free encyclopedia

Continuous integration describes a set of software engineering practices that speed up the delivery of software by decreasing integration times.

Theory

When embarking on a change, a developer takes a copy of the current code base on which to work. As changed code is submitted to the repository by other developers, this copy gradually ceases to reflect the repository code. When the developer submits code to the repository they must first update their code to reflect the changes in the repository since they took their copy. The more changes there are to the repository, the more work the developer must do before submitting their own changes.

Eventually, the repository may become so different from the developer's baseline that they enter what is sometimes called, "integration hell", where the time it takes to integrate is greater than the time it took to make their original changes. In a worst case scenario, the changes the developer is making may have to be discarded and the work redone.

Continuous Integration is the practice of integrating early and often, so as to avoid the pitfalls of "integration hell". The ultimate goal is to reduce timely rework and thus reduce cost and time. When done well, continuous integration has been shown to achieve these goals.

The rest of this article discusses best practice in how to achieve continuous integration, and how to automate this practice (automation is considered best practice itself).

Recommended Practices

Continuous integration itself refers to the practice of frequently integrating one's code with the code that is to be released (often this is the trunk, but that is not necessarily the case). The term “frequently” is open to interpretation, but is often taken to mean “many times every day.”

Maintain a code repository

This practice advocates the use of a revision control system for the project's source code. All artifacts that are needed to build the project should be placed in the repository. In this practice and in the revision control community, the convention is that the system should be buildable from a fresh checkout and not require additional dependencies. Extreme Programming advocate Martin Fowler also mentions that where branching is supported by tools, its use should be minimized. Instead, it is preferred that changes are integrated rather than creating multiple versions of the software that are maintained simultaneously. The mainline (or trunk) should be the place for the working version of the software.

Automate the build

Main article: Build automation

The system should be buildable using a single command. Many build tools exist, such as make, which has existed for many years. Other more recent tools like Ant, Maven, MSBuild or IBM Rational Build Forge are frequently used in **Continuous Integration** environments. Automation of the build should include automating the integration, which often includes deployment into a production-like environment. In many cases, the build script not only compiles binaries, but also generates documentation, website pages, statistics and distribution media (such as Windows MSI files or RPM files).

Make your build self-testing

This touches on another aspect of best practice, Test-driven development. Briefly, this is the practice of writing a test that demonstrates a lack of functionality in the system, and then writing the code that makes the test pass.
Once the code is built, all the tests should be run to confirm that it behaves as the developers expect it to behave.

Everyone commits every day

By committing regularly, every committer can reduce the number of conflicting changes. Checking in a week's worth of work runs the risk of conflicting with other features and can be very difficult to solve. Early, small conflicts in an area of the system cause team members to communicate about the change they are making.
Many programmers [1] recommend committing all changes at least once a day, and in addition performing a nightly build.

Every commit (to mainline) should be built

Commits to the current working version should be built to verify they have been integrated correctly. A common practice is to use Automated Continuous Integration, although this may be done manually. In fact, James Shore prefers this approach. For many, continuous integration is synonymous with using Automated Continuous Integration where a continuous integration server or daemon monitors the version control system for changes, then automatically runs the build process.

Keep the build fast

The build needs to be fast, so that if there is a problem with integration, it is quickly identified.

Test in a clone of the production environment

Having a test environment can lead to failures in tested systems when they are deployed to the production environment, because the production environment may differ from the test environment in a significant way.

Make it easy to get the latest deliverables

Making builds readily available to stakeholders and testers can reduce the amount of rework necessary when rebuilding a feature that doesn't meet requirements. Additionally, early testing reduces the chances that defects survive until deployment. Finding issues earlier also, in some cases, reduces the amount of work necessary to resolve them.

Everyone can see the results of the latest build

It should be easy to find out whether the build is broken and who made the change.

Automate Deployment

…

Extrait de **http://www.atlassian.com/software/bamboo/**

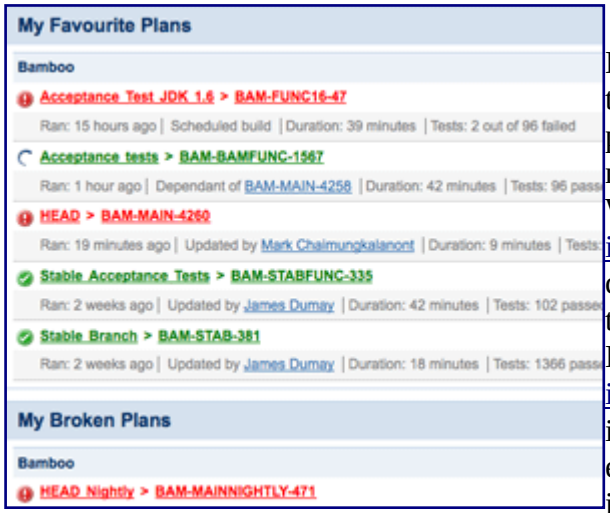
Bamboo - CI build server test automation test tools software testing continuous integration

Smarter feedback, better code!

Today's development teams are adopting continuous integration to increase productivity and improve code quality.
By automatically compiling and testing code as it changes, Bamboo provides instant feedback for developers and allows quick collaboration.

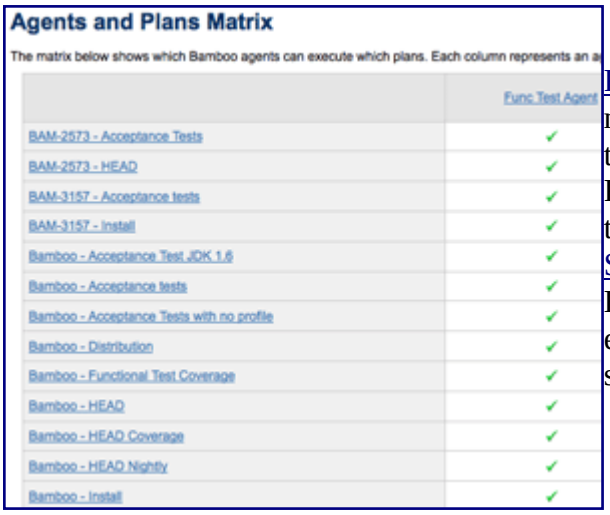
- **Instant scalability with Elastic Bamboo**

  Elastic Bamboo gives you ultimate flexibility by eliminating the need for dedicated hardware to scale your continuous integration environment, and allowing you to leverage Amazon Elastic Compute Cloud (EC2) with just a few clicks.
  Combine elastic agents in the cloud with local and remote agents on premises to keep your build queues short at all times through out the development cycle.
  Use the Bamboo Remote API to ramp agents up and down instantly. Pay only for what you use.



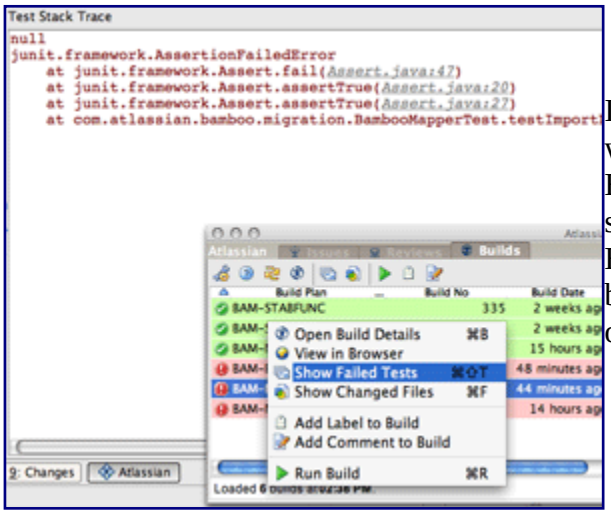
- **Continuous integration made easy**

Despite being a widely accepted best practice, many teams struggle to adopt a continuous integration process due to the perceived cost of setting up and maintaining such an environment.
With Bamboo, setting up your continuous integration process is simple. The installer auto-detects your development environment enabling you to start a build within minutes!
Bamboo's two-way notifications and intuitive web-interface provide developers with the necessary information and simple navigation to interact with each build as well as manage the entire continuous integration process.



- **Distribute builds across multiple machines**

Remote Bamboo agents allow builds to run across multiple machines and different platforms, extending test capabilities and maximizing build productivity. Remote agents are even able to restart themselves in the event of a crash with the Remote Agent Supervisor.
Establishing dependencies between build plans enables Bamboo to support even the most sophisticated continuous integration environments.



- **Continuous integration fits your environment**

Integration with your current tool set is vital which is why Bamboo is built with the developer in mind. Bamboo integrates with your favorite IDE and works seamlessly with JIRA, FishEye, Clover and Crowd. Bamboo's flexible plugin architecture allows you to build integration to any of the tools you use. Check out the extensive library of plugins available today.

Integrated Systems

SCM systems	Subversion, Perforce, CVS, *ClearCase*, *Mercurial*, *Dimension*, *Git*
Builders	Ant, Maven, Maven 2, Bash, MSBuilder, Visual Studio, Nant, *NoseXUnit*, custom scripts and command line builders (e.g. make), etc.
Testing frameworks	Any w/ JUnit XML output — including TestNG, Nunit, XUnit, MSTest, Nose, CppUnit, etc
Languages	Any — including Java, C, C++, .NET (C#, VB, etc.), Perl, PHP, Python, Ruby, etc.
Notifications	IM (Jabber or Google Talk), Email, RSS, Remote API
Atlassian tools	FishEye, Crowd, JIRA, Clover
IDE	IntelliJ IDEA, Eclipse (Beta)
Extensible plugins	**Build Metrics:** *Checkstyle, Crap4J, Cobertura, Coverage, FindBugs, JMeter, PMD, RCov, Simian* **Tools:** *Clean Test, Command Line, Project Graph, Test Threshold, Build Monitor, Disk Space Notification, Pre-Post Build Command* **Release Management & Deployment:** *Adhoc Builder, SFTP Publisher, Tag Build, Tagger, BuildBug, JiraVersions*

Note: User contributed, open source plugins noted in *italics*

Welcome to Buildbot!

Introduction

The BuildBot is a system to automate the compile/test cycle required by most software projects to validate code changes. By automatically rebuilding and testing the tree each time something has changed, build problems are pinpointed quickly, before other developers are inconvenienced by the failure. The guilty developer can be identified and harassed without human intervention. By running the builds on a variety of platforms, developers who do not have the facilities to test their changes everywhere before check in will at least know shortly afterwards whether they have broken the build or not. Warning counts, lint checks, image size, compile time, and other build parameters can be tracked over time, are more visible, and are therefore easier to improve.
The overall goal is to reduce tree breakage and provide a platform to run tests or code-quality checks that are too annoying or pedantic for any human to waste their time with. Developers get immediate (and potentially public) feedback about their changes, encouraging them to be more careful about testing before check in.
Features:

- run builds on a variety of slave platforms
- arbitrary build process: handles projects using C, Python, whatever
- minimal host requirements: python and Twisted
- slaves can be behind a firewall if they can still do checkout
- status delivery through web page, email, IRC, other protocols
- track builds in progress, provide estimated completion time
- flexible configuration by sub-classing generic build process classes
- debug tools to force a new build, submit fake Changes, query slave status
- released under the GPL
- History and Philosophy
- System Architecture
- Control Flow

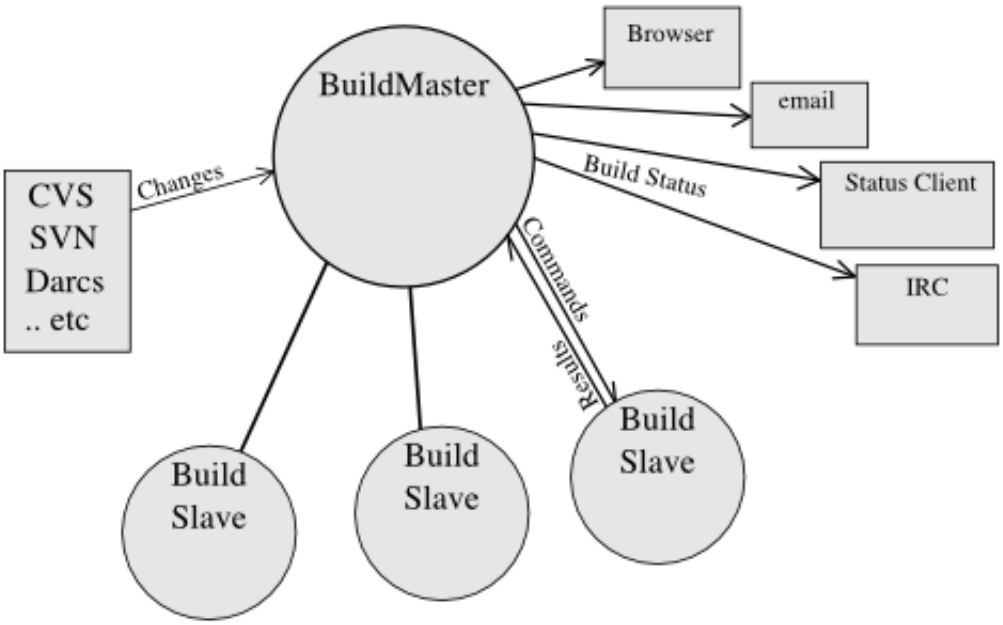
1.1 History and Philosophy

The Buildbot was inspired by a similar project built for a development team writing a cross-platform embedded system. The various components of the project were supposed to compile and run on several flavors of unix (Linux, Solaris, BSD), but individual developers had their own preferences and tended to stick to a single platform. From time to time, incompatibilities would sneak in (some unix platforms want to use string.h, some prefer strings.h), and then the tree would compile for some developers but not others. The buildbot was written to automate the human process of walking into the office, updating a tree, compiling (and discovering the breakage), finding the developer at fault, and complaining to them about the problem they had introduced. With multiple platforms it was difficult for developers to do the right thing (compile their potential change on all platforms); the buildbot offered a way to help.
Another problem was when programmers would change the behavior of a library without warning its users, or change internal aspects that other code was (unfortunately) depending upon. Adding unit tests to the code base helps here: if an application's unit tests pass despite changes in the libraries it uses, you can have more confidence that the library changes haven't broken anything. Many developers complained that the unit tests were inconvenient or took too long to run: having the buildbot run them reduces the developer's workload to a minimum.
In general, having more visibility into the project is always good, and automation makes it easier for developers to do the right thing. When everyone can see the status of the project, developers are encouraged to keep the tree in good working order. Unit tests that aren't run on a regular basis tend

to suffer from bit rot just like code does: exercising them on a regular basis helps to keep them functioning and useful.
The current version of the Buildbot is additionally targeted at distributed free-software projects, where resources and platforms are only available when provided by interested volunteers. The buildslaves are designed to require an absolute minimum of configuration, reducing the effort a potential volunteer needs to expend to be able to contribute a new test environment to the project. The goal is for anyone who wishes that a given project would run on their favorite platform should be able to offer that project a buildslave, running on that platform, where they can verify that their portability code works, and keeps working.

1.2 System Architecture

The Buildbot consists of a single buildmaster and one or more buildslaves, connected in a star topology. The buildmaster makes all decisions about what, when, and how to build. It sends commands to be run on the build slaves, which simply execute the commands and return the results. (Certain steps involve more local decision making, where the overhead of sending a lot of commands back and forth would be inappropriate, but in general the buildmaster is responsible for everything).
The buildmaster is usually fed Changes by some sort of version control system (see Change Sources), which may cause builds to be run. As the builds are performed, various status messages are produced, which are then sent to any registered Status Targets (see Status Delivery).

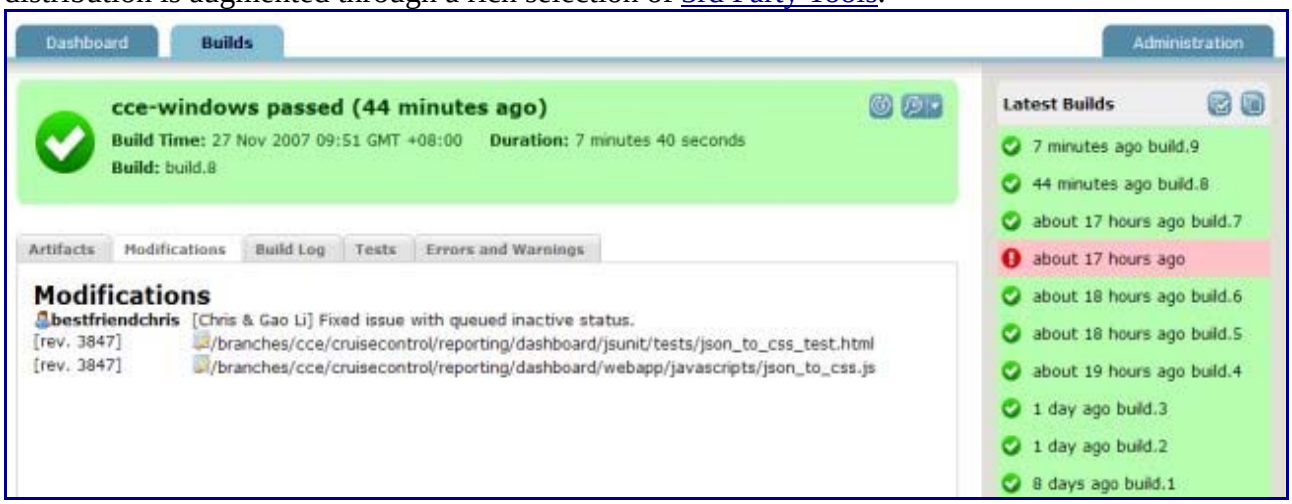


The buildmaster is configured and maintained by the “buildmaster admin”, who is generally the project team member responsible for build process issues. Each buildslave is maintained by a “buildslave admin”, who do not need to be quite as involved. Generally slaves are run by anyone who has an interest in seeing the project work well on their favorite platform.

Extrait de http://cruisecontrol.sourceforge.net/ et de http://cruisecontrol.sourceforge.net/overview.htm

CruiseControl

CruiseControl is both a continuous integration tool and an extensible framework for creating a custom continuous build process. It includes dozens of plugins for a variety of source controls, build technologies, and notifications schemes including email and instant messaging. A web interface provides details of the current and previous builds. And the standard CruiseControl distribution is augmented through a rich selection of 3rd Party Tools.



CruiseControl is written in Java but is used on a wide variety of projects. There are builders supplied for Ant, NAnt, Maven, Phing, Rake, and Xcode, and the catch-all exec builder that can be used with any command-line tool or script.
CruiseControl is open source software and is developed and maintained by a group of dedicated volunteers. CruiseControl is distributed under a BSD-style license.

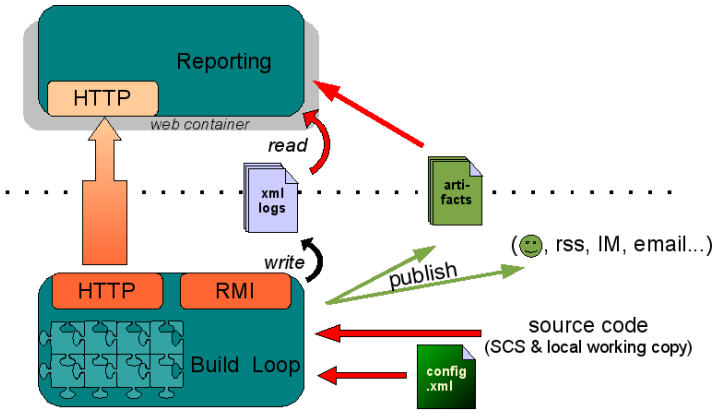
Overview

CruiseControl is composed of 3 main modules:

- the build loop: core of the system, it triggers build cycles then notifies various listeners (users) using various publishing techniques. The trigger can be internal (scheduled or upon changes in a SCM) or external. It is configured in a xml file which maps the build cycles to certain tasks, thanks to a system of plugins. Depending on configuration, it may produce build artifacts.
- the jsp reporting application allows the users to browse the results of the builds and access the artifacts
- the dashboard provides a visual representation of all project build statuses.

This modularity allows users to install CruiseControl where it will best fit their needs and environment.

Using remoting technologies (HTTP, RMI), it is possible to control and monitor the CruiseControl build loop. Those are turned off by default for obvious security reasons.



CruiseControl can be installed from source, or using the all in one binary installation.

The CruiseControl Build Loop

CruiseControl runs a Build Loop, which is designed to run as a daemon process which will periodically check your source control tool for changes to your code base, build if necessary, and send out a notification regarding the status of the build.

This is a loop that you define in the config XML file by defining project(s) containing information about timing, logging, and notification information (see ConfigFiles). You can have interested parties notified by e-mail, or use the Reporting Application that comes with CruiseControl to make the build and testing results available, or both. There are also many other so-called Publishers available, and you can easily write your own.

You can update a project with new rules/information which will be picked up by the loop every time it goes around, depending on your needs. You can define multiple projects in the same config file to run (and compile & test) in the same CruiseControl instance. The default is to use a build queue that builds one project at a time, but if you're using a CruiseControl with a version higher than 2.1.6 you can also have projects building in parallel, should you want that configuration. Just keep in mind that you always need to restart CruiseControl if you've added or removed projects, since it will only pick up changes to existing projects on every loop.

The build loop depends on Ant, jakarta's simple, extensible, XML-formatted version of make which is the de-facto standard for java build tools. Once you have your project building with Ant, setting up and using cruise control is twice as easy. You can also use Maven, which is becoming an increasingly popular alternative to Ant.

In addition, because Ant or Maven still allow you to execute legacy makefiles and scripts, CruiseControl only magnifies and enhances the longevity of any dependable, mature build processes you may already have.

Distributed extensions

The distributed package is in the process of being merged into the core product.

Introduction

This "distributed" contrib package for Cruise Control allows a master build machine to distribute build requests to other physical machines on which the builds are performed and to return the results to the master.

In order to extend Cruise Control without requiring that our distributed extensions be merged in with the core Cruise Control code, we decided to add our code as a new contrib package. This complicates configuration a bit, but carefully following the following steps should have you distributing builds in no time. You should, however, already be familiar with Cruise Control if you expect to succeed with this more complex arrangement.

Overview

The distributed extensions make use of Jini for the service lookup and RMI features it provides. In addition to the usual Cruise Control startup the user will have to start up a Jini service registrar and HTTP class server. Also, each build agent machine will need to have code installed locally and will need to start up and register their availability with the registrar. Once a federation of one or more agents is registered with a running registrar, Cruise Control has the ability to distribute builds through a new Distributed Master Builder that wraps an existing Builder in the CC configuration file. Examples are given below. Doing distributed builds is seamless in Cruise Control and the user has the option of only distributing builds for projects they choose to distribute.