TLB

rocket booster for your build

Pavan KS[1] Janmejay Singh[2]

[1]mail: itspanzi@gmail.com
blog: http://itspanzi.blogspot.com

[2]mail: singh.janmejay@gmail.com
blog: http://codehunk.wordpress.com

RubyConf India 2011, Bangalore

TLB what?

- A Free and Open Source software: *BSD licensed*
- Cuts your build time by executing tests parallely on a grid
- Supports multiple build tools
- Supports multiple testing tools
- Supports multiple languages

Whats in it for me?

What can I expect out of this session?

- A small, hopefully interesting, story that describes *a problem*
- How we solved *that problem*
- How we can help you solve it
- How you can help us, help others solve it!

Outline

Outline

The Story

This is the story of how we went from...

build

Run: 1 of 1 | Building | Triggered: by anonymous at 2010-08-06T18:55:57+05:30 Duration: In Progress |

Overview | Pipeline Dependencies | Materials | Jobs | Tests

Name	Result	State	Duration	Agent
firefox	Active	Building	49 minutes and 9 seconds	blrstdcrspbs06.thoughtworks.com(10.4.7.86)

What is it doing that takes more than 49 minutes?

This is the story of how we went from...

build

Run: 1 of 1 | Building | Triggered: by anonymous at 2010-08-06T18:55:57+05:30 | Duration: In Progress

Overview | Pipeline Dependencies | Materials | Jobs | Tests

Name	Result	State	Duration	Agent
firefox	Active	Building	49 minutes and 9 seconds	blrstdcrspbs06.thoughtworks.com(10.4.7.86)

What is it doing that takes more than 49 minutes?

to...

build

Run: 1 of 1 Passed | Triggered: by anonymous at 2010-08-06T18:35:50+05:30 Duration: 00:18:15 |

Overview | Pipeline Dependencies | Materials | Jobs | Tests

Name	Result	State	Duration	Agent
firefox-1	Passed	Completed	11 minutes and 32 seconds	blrstdcrspbs05.thoughtworks.com(10.4.7.85)
firefox-2	Passed	Completed	7 minutes and 40 seconds	blrstdcrspbs03.thoughtworks.com(10.4.7.83)
firefox-3	Passed	Completed	7 minutes and 57 seconds	blrstdcrspbs02.thoughtworks.com(10.4.7.82)
firefox-4	Passed	Completed	17 minutes and 58 seconds	blrstdcrspbs04.thoughtworks.com(10.4.7.84)
firefox-5	Passed	Completed	6 minutes and 14 seconds	blrstdcrspbs07.thoughtworks.com(10.4.7.87)
firefox-6	Passed	Completed	2 minutes and 4 seconds	blrstdcrspbs01.thoughtworks.com(10.4.7.81)
rails	Passed	Completed	1 minute and 23 seconds	blrstdcrspbs08.thoughtworks.com(10.4.7.88)

to...

build

Run: 1 of 1 Passed | Triggered: by cruise at 2010-08-06T15:11:26+05:30 Duration: 00:09:39 |

Overview | Pipeline Dependencies | Materials | Jobs | Tests

Name	Result	State	Duration	Agent
firefox-1	Passed	Completed	9 minutes and 22 seconds	blrstdcrspbs04.thoughtworks.com(10.4.7.84)
firefox-2	Passed	Completed	8 minutes and 58 seconds	blrstdcrspbs05.thoughtworks.com(10.4.7.85)
firefox-3	Passed	Completed	9 minutes and 4 seconds	blrstdcrspbs07.thoughtworks.com(10.4.7.87)
firefox-4	Passed	Completed	8 minutes and 50 seconds	blrstdcrspbs08.thoughtworks.com(10.4.7.88)
firefox-5	Passed	Completed	8 minutes and 36 seconds	blrstdcrspbs03.thoughtworks.com(10.4.7.83)
firefox-6	Passed	Completed	8 minutes and 51 seconds	blrstdcrspbs02.thoughtworks.com(10.4.7.82)
rails	Passed	Completed	1 minute and 18 seconds	blrstdcrspbs01.thoughtworks.com(10.4.7.81)

- With just a few lines changed in the build script & executing tests parallely.
- The longish build 'firefox' split into 6 partitions to executed parallely on 6 physical machines.

- With just a few lines changed in the build script & executing tests parallely.
- The longish build 'firefox' split into 6 partitions to executed parallely on 6 physical machines.

Outline

- Fast build = Rapid development
- Devs spend less time waiting to checkin
  - Need not be limited to unit/integration tests
  - Functional/Acceptance tests
- Easier for devs to run precommit builds - Results in pulling upstream changes often and running builds frequently

- Fast build = Rapid development
- Devs spend less time waiting to checkin
  - Need not be limited to unit/integration tests
  - Functional/Acceptance tests
- Easier for devs to run precommit builds - Results in pulling upstream changes often and running builds frequently

- Fast build = Rapid development
- Devs spend less time waiting to checkin
  - Need not be limited to unit/integration tests
  - Functional/Acceptance tests
- Easier for devs to run precommit builds - Results in pulling upstream changes often and running builds frequently

Also we know that...

- Major part of build time is spent in running tests
- Speeding up builds is non trivial
- Most teams have a dev task for this don't they??

Also we know that...

- Major part of build time is spent in running tests
- Speeding up builds is non trivial
- Most teams have a dev task for this don't they??

Also we know that...

- Major part of build time is spent in running tests
- Speeding up builds is non trivial
- Most teams have a dev task for this don't they??

Common solutions...

- Split applications into modules
  - Difficult to model (typically end up in diamond dependencies)
  - Pipelines (unit -> integration -> smoke -> functional) (serial process)
  - BUT, if downstream dependencies fail, turn around time to fix is huge
- Throw more hardware at it - Slice and dice
  - Hand written partitioning using directories/tags etc (unequal partitions)

Logical! but Suboptimal :-(
Nither too efficient nor effective.

Common solutions...

- Split applications into modules
  - Difficult to model (typically end up in diamond dependencies)
  - Pipelines (unit -> integration -> smoke -> functional) (serial process)
  - BUT, if downstream dependencies fail, turn around time to fix is huge
- Throw more hardware at it - Slice and dice
  - Hand written partitioning using directories/tags etc (unequal partitions)

Logical! but Suboptimal :-(
Nither too efficient nor effective.

Common solutions...

- Split applications into modules
  - Difficult to model (typically end up in diamond dependencies)
  - Pipelines (unit -> integration -> smoke -> functional) (serial process)
  - BUT, if downstream dependencies fail, turn around time to fix is huge
- Throw more hardware at it - Slice and dice
  - Hand written partitioning using directories/tags etc (unequal partitions)

Logical! but Suboptimal :-(
Nither too efficient nor effective.

Common solutions...

- Split applications into modules
  - Difficult to model (typically end up in diamond dependencies)
  - Pipelines (unit -> integration -> smoke -> functional) (serial process)
  - BUT, if downstream dependencies fail, turn around time to fix is huge
- Throw more hardware at it - Slice and dice
  - Hand written partitioning using directories/tags etc (unequal partitions)

Logical! but Suboptimal :-(
Nither too efficient nor effective.

Common solutions...

- Split applications into modules
  - Difficult to model (typically end up in diamond dependencies)
  - Pipelines (unit -> integration -> smoke -> functional) (serial process)
  - BUT, if downstream dependencies fail, turn around time to fix is huge
- Throw more hardware at it - Slice and dice
  - Hand written partitioning using directories/tags etc (unequal partitions)

Logical! but Suboptimal :-(
Nither too efficient nor effective.

Common solutions...

- Split applications into modules
  - Difficult to model (typically end up in diamond dependencies)
  - Pipelines (unit -> integration -> smoke -> functional) (serial process)
  - BUT, if downstream dependencies fail, turn around time to fix is huge
- Throw more hardware at it - Slice and dice
  - Hand written partitioning using directories/tags etc (unequal partitions)

Logical! but Suboptimal :-(
Nither too efficient nor effective.

Common solutions...

- Split applications into modules
  - Difficult to model (typically end up in diamond dependencies)
  - Pipelines (unit -> integration -> smoke -> functional) (serial process)
  - BUT, if downstream dependencies fail, turn around time to fix is huge
- Throw more hardware at it - Slice and dice
  - Hand written partitioning using directories/tags etc (unequal partitions)

Logical! but Suboptimal :-(

Nither too efficient nor effective.

Common solutions...

- Split applications into modules
  - Difficult to model (typically end up in diamond dependencies)
  - Pipelines (unit -> integration -> smoke -> functional) (serial process)
  - BUT, if downstream dependencies fail, turn around time to fix is huge
- Throw more hardware at it - Slice and dice
  - Hand written partitioning using directories/tags etc (unequal partitions)

Logical! but Suboptimal :-(
Nither too efficient nor effective.

Optimal solution

Optimal solution can be obtained by minimizing, the following expression.

$$D(A_1, A_2, ...A_n) = max\left\{\sum_{x \in A_i} x\right\} - min\left\{\sum_{x \in A_j} x\right\} \quad (1)$$

Its ideal when global minima for this function is reached.

Outline

What if partitioning can be off-loaded?

What does TLB do?

- Makes `n` partitions
- Understands which partition the *current test runner process* is
- Each partition runs only `one` of the `n` *mutually exclusive & collectively exhaustive* sets

What if partitioning can be off-loaded?

What does TLB do?

- Makes `n` partitions
- Understands which partition the *current test runner process* is
- Each partition runs only `one` of the `n` *mutually exclusive & collectively exhaustive* sets

What if partitioning can be off-loaded?

What does TLB do?

- Makes `n` partitions
- Understands which partition the *current test runner process* is
- Each partition runs only `one` of the `n` *mutually exclusive & collectively exhaustive* sets

For example:

Some potential balancing strategies could be

- Partition tests to make every set have equal number of tests
- Or to have every set take about the same time to finish

Some of these strategies require a central place to store and retrive test-data(running-time, result etc)

For example:

Some potential balancing strategies could be

- Partition tests to make every set have equal number of tests
- Or to have every set take about the same time to finish

Some of these strategies require a central place to store and retrive test-data(running-time, result etc)

For example:

Some potential balancing strategies could be

- Partition tests to make every set have equal number of tests
- Or to have every set take about the same time to finish

Some of these strategies require a central place to store and retrive test-data(running-time, result etc)

For example:

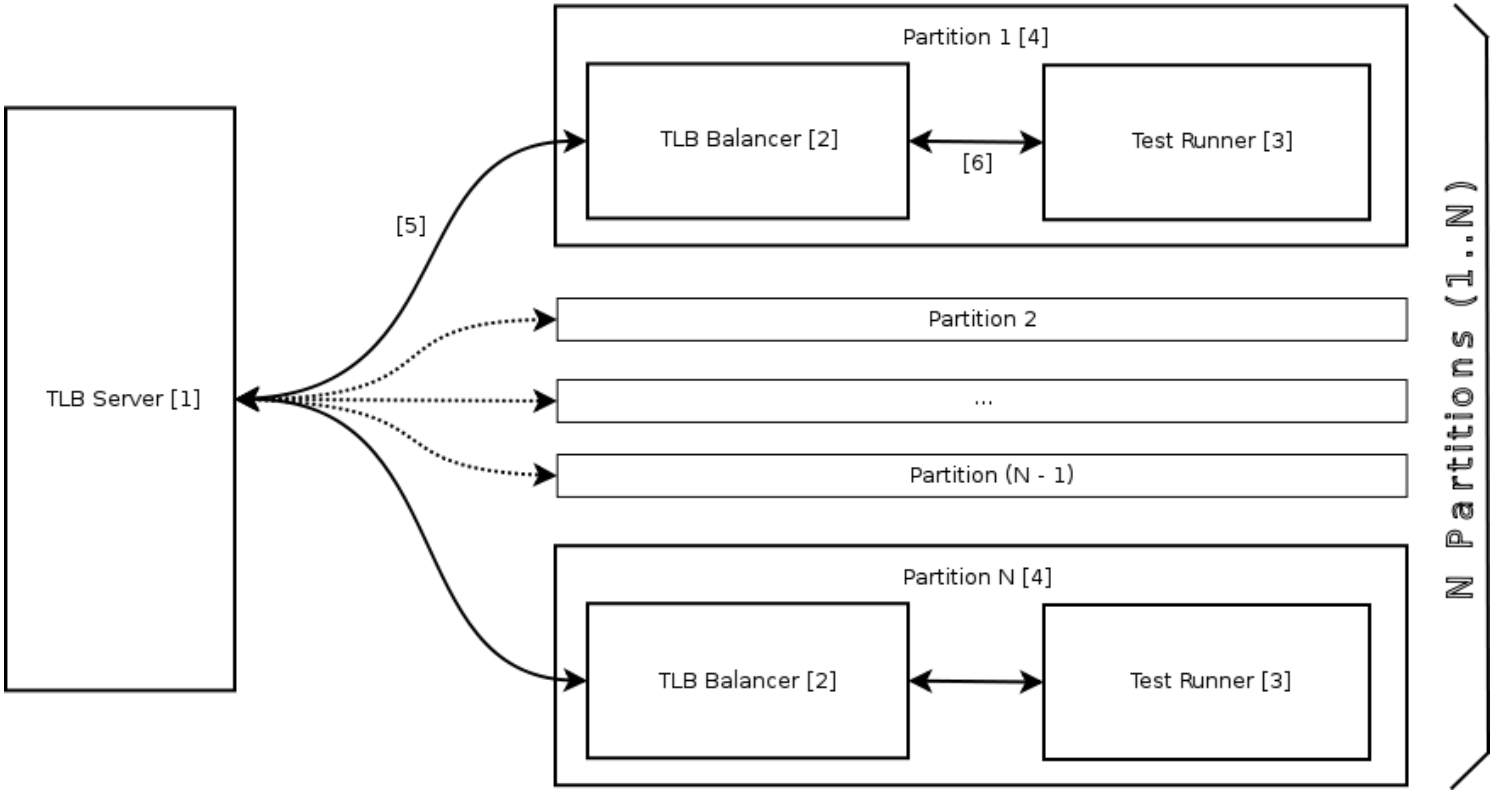
Some potential balancing strategies could be

- Partition tests to make every set have equal number of tests
- Or to have every set take about the same time to finish

Some of these strategies require a central place to store and retrive test-data(running-time, result etc)

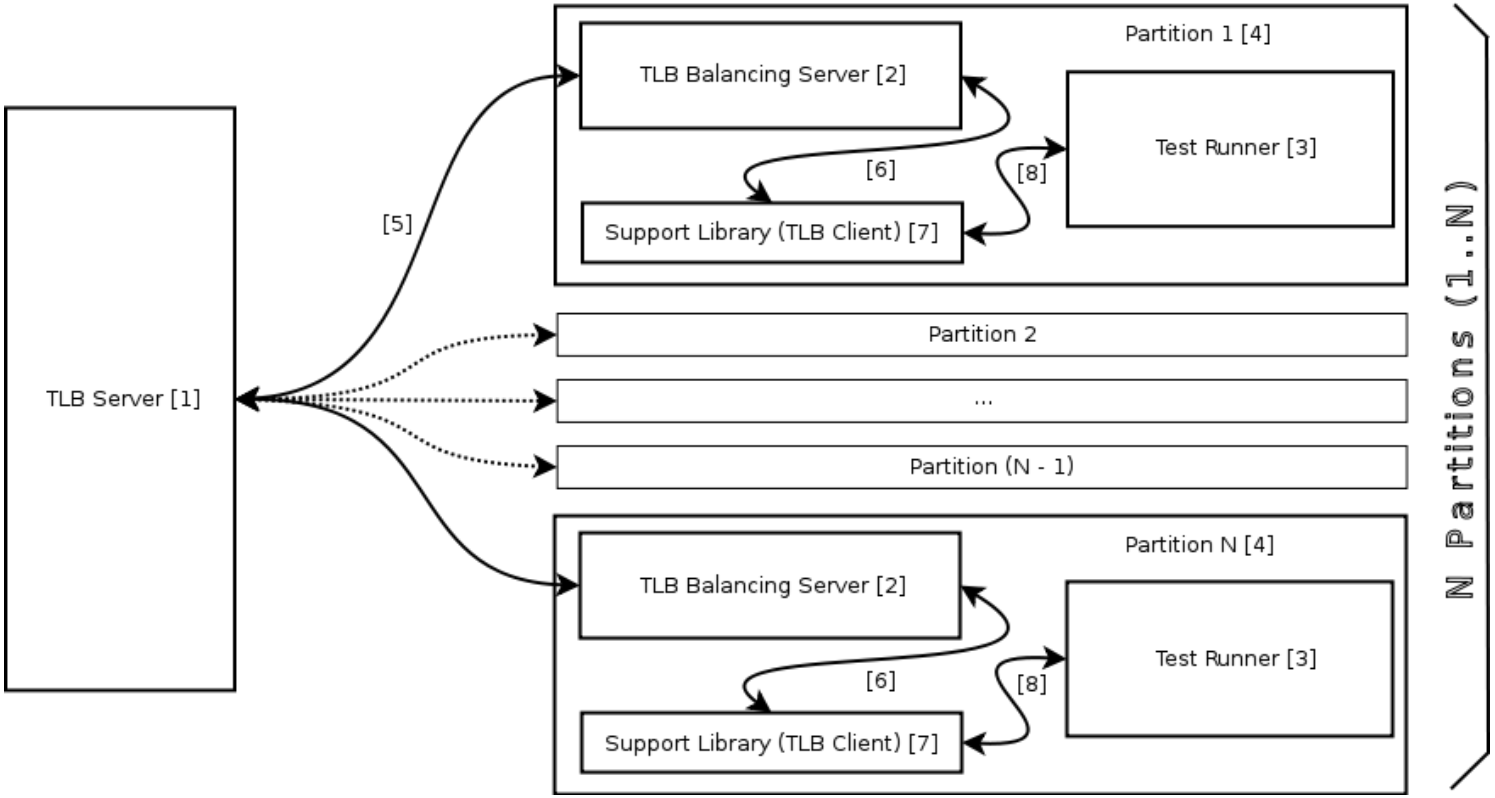
Setup

Typical Setup



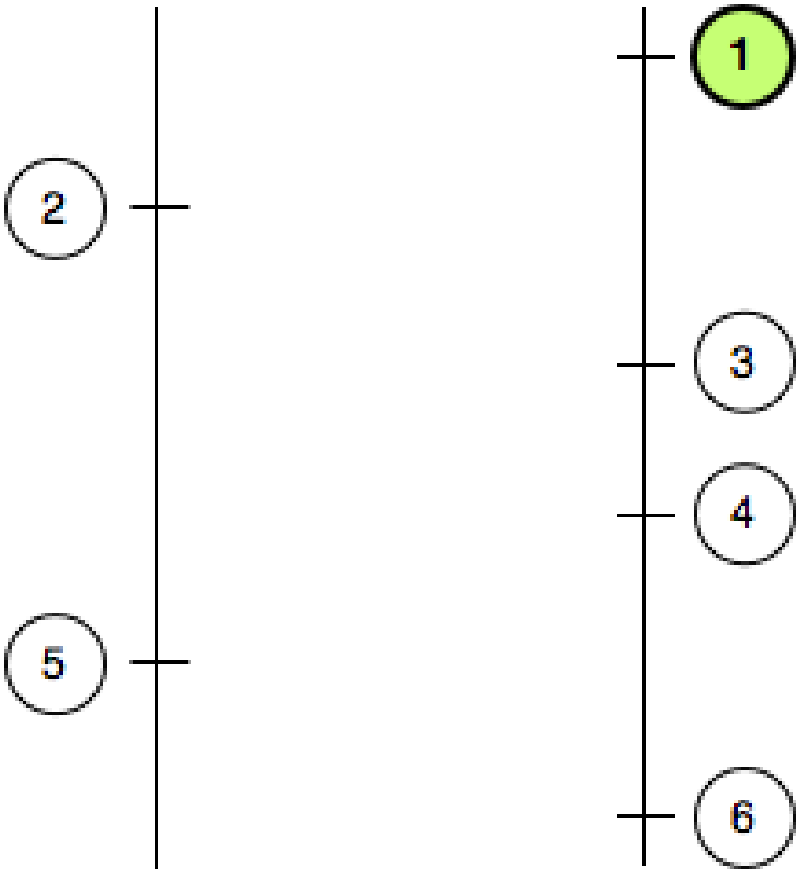
Setup

Alien Environment Setup



Communication

Server-Client-Runner Talk

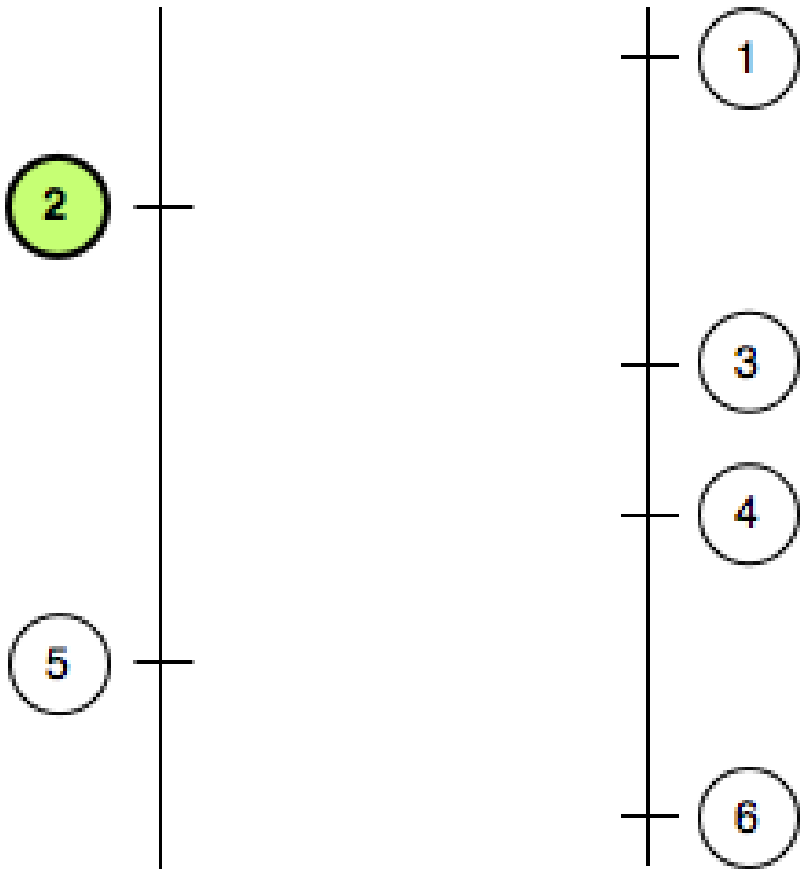


Step 1

Receive list of to-be-run tests from the testing-framework

Communication

Server-Client-Runner Talk

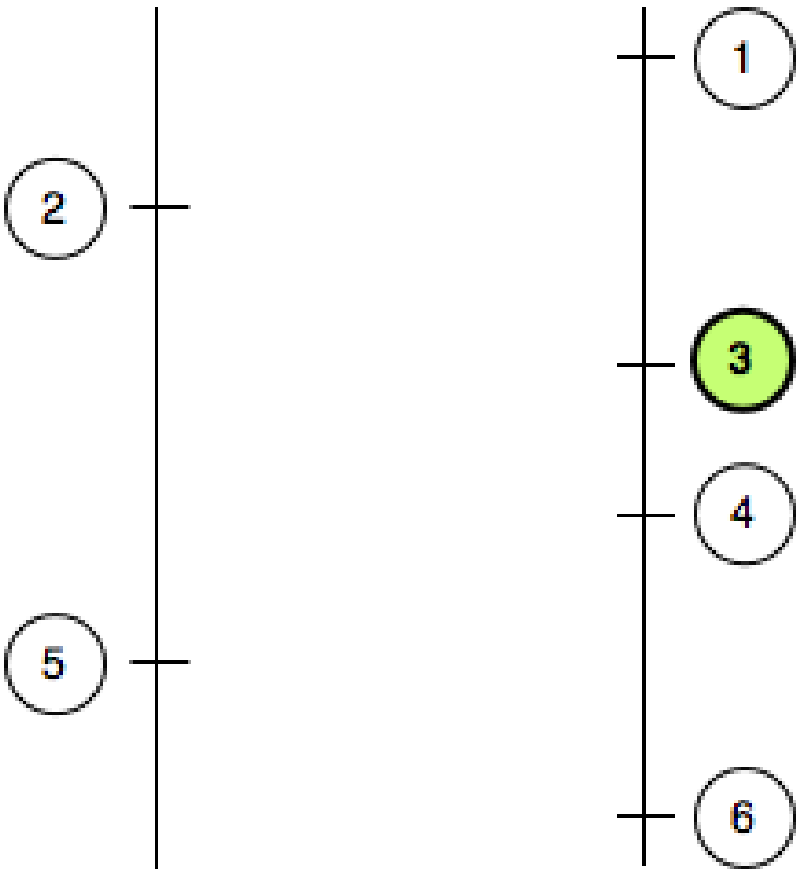


Step 2

Fetch historical test data from TLB server(tests that failed in the previous run/time taken by each test)

Communication

Server-Client-Runner Talk

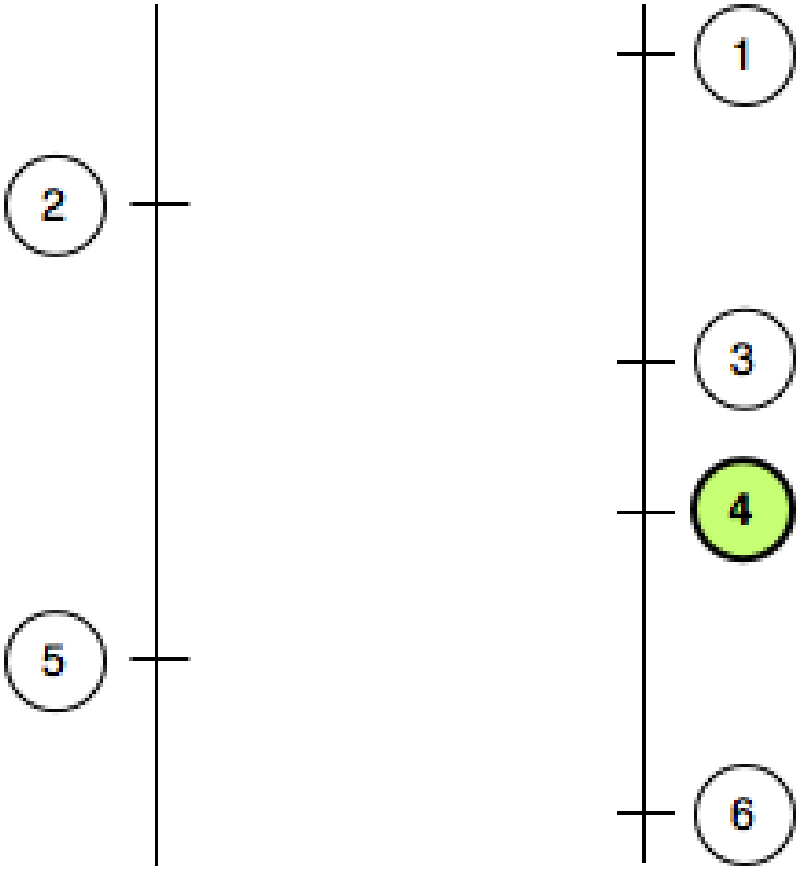


Step 3

- Prune the list of to-be-run tests to get tests to be actually executed(other partitions take care of pruned items)
- Re-order the pruned list,for instance *pull tests that failed in the previous run to execute first*

Communication

Server-Client-Runner Talk

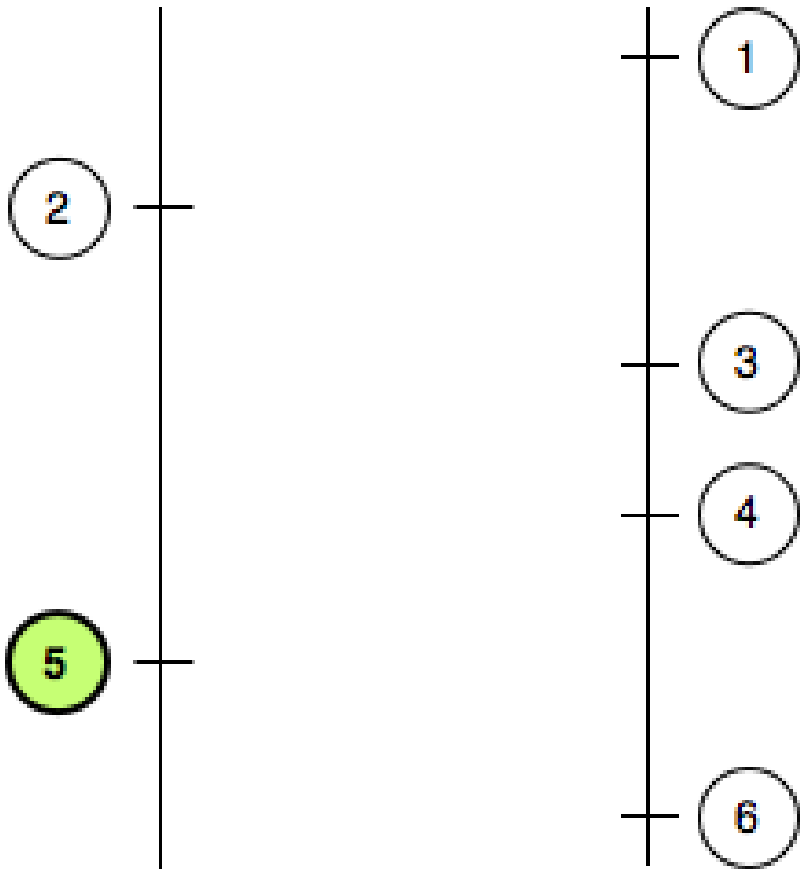


Step 4

- Let the party begin, *execute the pruned tests*
- Continue capturing test-result/test-time as the suites run

Communication

Server-Client-Runner Talk

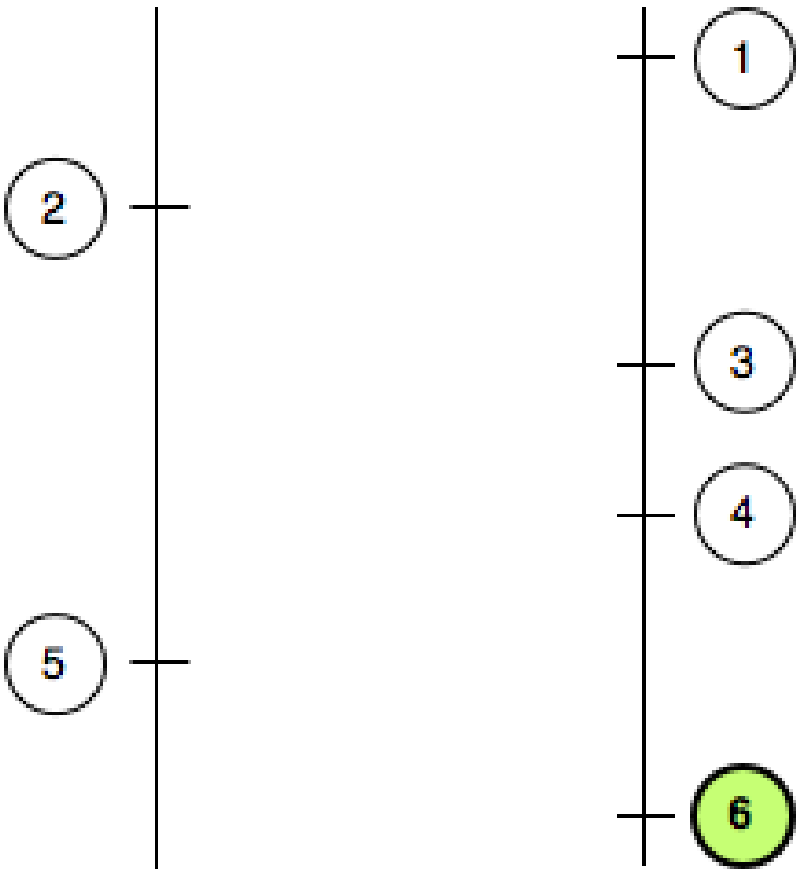


Step 5

Some feedback to the server; post test-run-time/test-results back, seeding data for future runs

Communication

Server-Client-Runner Talk



Step 6

Terminate gracefully; build task returns

Ok enough of handwaving!

Well, that was all too abstract, lets see what TLB has to offer in terms of concrete features.

Outline

TLB Client

the workhorse

TLB client has two major sub-units.

- Balancer - *the pruner guy* (chosen by setting an environment variable TLB_CRITERIA)
- Orderer - *the shuffler guy* (chosen by setting an environment variable TLB_ORDERER)

both environment variables require fully qualified java class-names

TLB Client

the workhorse

TLB client has two major sub-units.

- Balancer - *the pruner guy* (chosen by setting an environment variable TLB_CRITERIA)
- Orderer - *the shuffler guy* (chosen by setting an environment variable TLB_ORDERER)

both environment variables require fully qualified java class-names

TLB Client

the workhorse

TLB client has two major sub-units.

- Balancer - *the pruner guy* (chosen by setting an environment variable TLB_CRITERIA)
- Orderer - *the shuffler guy* (chosen by setting an environment variable TLB_ORDERER)

both environment variables require fully qualified java class-names

TLB Client

Balancer

Count-based Balancing

20 tests / 4 splits = 5 tests on each

- Conceptually straight-forward
- Inefficient in practice
- TLB uses this as a fallback, not recomended as preffered algorithm

TLB Client

Balancer

Time-based Balancing

(inspired by Amdahl's law)

N tests / 4 splits $\approx$ 4 splits that take equal time

- Much better, yields fairly close to ideal solution
- One slow machine can not only slow down the current run, but skew balancing on the next one too

TLB Client

Balancer

Smoothened Time-based Balancing

(Ensures no outliers, builds on top of time based balancing)

N tests / 4 splits $\approx$ 4 splits which take equal time based on history over past few/several runs

While exponential smoothing, every test-time entry $S_t \quad \forall t > 0$ is recorded as:

$$S_1 = x_o \tag{2}$$

$$S_t = \alpha x_{t-1} + (1 - \alpha) S_{t-1} \quad \forall \begin{cases} t > 1 \\ 0 < \alpha < 1 \end{cases} \tag{3}$$

Where α is the factor of smoothing, which can be tuned externally for every partition and x is unsmoothed reading.

TLB Client

Balancer

Default-chain Balancing

- Allows users to define *criteria chain*, which is a COLON(:) seperated list of algorithms
- The chain may include some custom balancer recepies of your own
- Used to ensure build doesn't fail when no data available to do advanced algorithms like Time-balance
- Allows defaulting to simpler algorithms like Count-balancing

TLB Client

Balancer

XYZ Balancing

(This is not a canned algorithm, its something you can create)

- You can create your custom balancing algorithm, and use it with TLB
- The contract is enforced by a java abstract class called *TestSplitterCriteria*
- If it can potentially fail in some situations and you want a fallback, you can use *DefaultingTestSplitterCriteria* with your balancer in chain
- Note: Algorithm need to be repeatable, since its executed on every partition. *Mutual-exclusion & Collective-exhausion* are imparative.

TLB Client

Orderer

Failed First Orderer

(Runs tests that failed last time around, first)

- Perfect for fixing builds that have a tendency to break after 6 in the evening
- You don't need to wait for the entire build, just watch the console log for a few minutes, as you see the test you fixed pass and scroll by

TLB Client

Orderer

ABC Orderer

(This is not a canned algorithm, its something you can create)

- You can create your custom ordering algorithm, and use it with TLB
- The contract is enforced by a java abstract class called *TestOrderer*
- Ordering tests to ensure execution-order/side-effects is a slippery slope and is considered an *ANTI PATTERN*, so we strongly recomend not abusing Ordering facility

Outline

TLB Client needs to embed in build environment and interact with a testing-framework

- Balancing JUnit test-suite using Apache-Ant
- Balancing RSpec test-suite using Rake

TLB Client needs to embed in build environment and interact with a testing-framework

- Balancing JUnit test-suite using Apache-Ant
- Balancing RSpec test-suite using Rake

TLB Client needs to embed in build environment and interact with a testing-framework

- Balancing JUnit test-suite using Apache-Ant
- Balancing RSpec test-suite using Rake

Outline

Leverage parallel execution capabilities of

tools like...

or for that matter

Hudson, Bamboo, TeamCity, Ant Hill Pro(if you are rich enough),or even Capistrano/Shell script fork(if you are a poor dev like us).

Supported Setup(s)

When we say Ruby, we mean both 1.9 and 1.8

- Rspec 1.x & 2.x using Rake on MRI and JRuby
- Test::Unit using Rake on MRI and JRuby
- Cucumber using Rake on MRI and JRuby
- Junit using Ant or Buildr for Java
- Twist using Ant or Buildr for Java

Supported Setup(s)

When we say Ruby, we mean both 1.9 and 1.8

- Rspec 1.x & 2.x using Rake on MRI and JRuby
- Test::Unit using Rake on MRI and JRuby
- Cucumber using Rake on MRI and JRuby
- Junit using Ant or Buildr for Java
- Twist using Ant or Buildr for Java

Supported Setup(s)

When we say Ruby, we mean both 1.9 and 1.8

- Rspec 1.x & 2.x using Rake on MRI and JRuby
- Test::Unit using Rake on MRI and JRuby
- Cucumber using Rake on MRI and JRuby
- Junit using Ant or Buildr for Java
- Twist using Ant or Buildr for Java

Supported Setup(s)

When we say Ruby, we mean both 1.9 and 1.8

- Rspec 1.x & 2.x using Rake on MRI and JRuby
- Test::Unit using Rake on MRI and JRuby
- Cucumber using Rake on MRI and JRuby
- Junit using Ant or Buildr for Java
- Twist using Ant or Buildr for Java

Supported Setup(s)

When we say Ruby, we mean both 1.9 and 1.8

- Rspec 1.x & 2.x using Rake on MRI and JRuby
- Test::Unit using Rake on MRI and JRuby
- Cucumber using Rake on MRI and JRuby
- Junit using Ant or Buildr for Java
- Twist using Ant or Buildr for Java

Working on...

Junit using Maven on Java

Waiting on http://jira.codehaus.org/browse/SUREFIRE-726

We plan to support...

- TestNG on Java
- JBehave on Java
- NUnit on .Net
- MS Test on .Net
- PyUnit on Python
- CPPUnit on C++
- 5am (fiveam) on CommonLisp
- ... on ...
- NAnt on .Net
- MS Build on .Net
- ... on ...

While thats our wish-list

- Bad news is, we haven't started work on most of these yet.
- Good news is, we have good hackers, like yourself, listening to us here, who can help!

We'd love to support anything else that you can make time to contribute :-)

While thats our wish-list

- Bad news is, we haven't started work on most of these yet.
- Good news is, we have good hackers, like yourself, listening to us here, who can help!

We'd love to support anything else that you can make time to contribute :-)

While thats our wish-list

- Bad news is, we haven't started work on most of these yet.
- Good news is, we have good hackers, like yourself, listening to us here, who can help!

We'd love to support anything else that you can make time to contribute :-)

Outline

Partitioning Approaches

- Greedy Algorithm (time-balancing-criteria uses this one)
- Differencing Algorithm (half done, not upstrem yet)
- GA spike, terrifingly good (upstream as a spike, written in CommonLisp, needs some more tuning)

GA spike for set-partitioning

TLB doesn't have it yet; coming soon!

Check details @ http://github.com/test-load-balancer/set-part

Keep children when..

fitness function

The new generation

$$A'_x = x_1 \ldots x_i + y_j \ldots y_{j+\delta'} + x_{j+\delta+1} \ldots x_n \tag{4}$$

$$A'_y = y_1 \ldots y_j + x_i \ldots x_{i+\delta} + y_{j+\delta'+1} \ldots y_m \tag{5}$$

Can be considered better than parents

$$A_x \quad and \quad A_y \quad iff:$$

$$SD(A_x, A_y) > SD(A'_x, A'_y) \left\{ \bar{A} = \frac{\sum_{i=1}^{X} U_{x_i}}{k} \quad : \quad k = no. \quad of \quad bags \right. \tag{6}$$

This is just one way to mesure the evolution of the new generation.

Random numbers

0 < n < 10000; 10 generations

```
NIL
CL-USER> (time (prt-spk::do-generations 10 nil 10 1000 10000))
before: (5021564 4902899 5092426 4907967 5034736 4983726 5043880 4855970
         4960732 4904700)(max-min-deviation: 236456)
after: (4932643 5011567 4957064 4981376 4963501 4974537 4969921 4974010
        4970827 4973154)(max-min-deviation: 78924)
Evaluation took:
  0.007 seconds of real time
  0.005999 seconds of total run time (0.004999 user, 0.001000 system)
  85.71% CPU
  18,116,545 processor cycles
  1,875,552 bytes consed
1-U:**-  *slime-repl sbcl*   Bot (586,9)    (REPL Autodoc mate)----------
(do-generations generations &optional verbose (number-of-bags 10) (bags-of-size 10
```

Random numbers

0 < n < 10000; 100 generations

```
NIL
CL-USER> (time (prt-spk::do-generations 100 nil 10 1000 10000))
before: (5094937 4951008 4938120 4913776 4924320 4902091 5011473 5097288
         4957150 4946717)(max-min-deviation: 195197)
after: (4982679 4981543 4983217 4984007 4983304 4983944 4983494 4983874
        4973524 4973184)(max-min-deviation: 1864)
Evaluation took:
  0.059 seconds of real time
  0.008990 seconds of total run time (0.057991 user, 0.000999 system)
  [ Run times con ist of 0.012 seconds GC time, and 0.047 second  non-GC time. ]
  100.00% CPU
  155,563,070yprocessor cycles
NIL3,404,1 4 bytes consed
NIL                          o-generations 100 nil 10 1000 10000))
CL-USER>
1-U:**-  *slime-repl sbcl*   Bot (599,9)    (REPL Autodoc mate)-------------------------
```

Random numbers

0 < n < 10000; 1000 generations

```
NIL                          o-generations 100 nil 10 1000 10000))
CL-USER> (time (prt-spk::do-generations 1000 nil 10 1000 10000)) 5097288
before: (5092036 5062438 5013295 5108255 5035299 5136415 4961062 4925085
       ( 9883012 4981985)(max-min-deviation: 111330)
after: (5039657 5040030 5039818 5039941 5039860 5039928 5039898 5039928
      + 5039903 5039919)(max-min-deviation: 373)
Evaluation took:
  0.514 seconds of real time
  0.513922 seconds of total run time (0.508923 user, 0.004999 system)
  [ Run times consist of 0.042 seconds GC time, and 0.472 seconds non-GC time. ]
  100.00% CPU
  1,368,348,202 processor cycles
  132,044,224 bytes consed
NIL
CL-USER>
1-U:**-  *slime-repl sbcl*   Bot (612,9)    (REPL Autodoc mate)----------------------
```

Random numbers

0 < n < 10000; 10000 generations

```
CL-USER> (time (prt-spk::do-generations 10000 nil 10 1000 10000)) 925085
before: (4993636 5127955 5014096 4995674 5072414 5090845 4960698 5033370
       ( 503155750436306)(max-min-deviation: 175608)
after: (5045649 5045981 5045649 5045672 5045651 5045668 5045653 5045666
      t on to55:     57)(max-min-deviation: 32)
Evaluation took:
  5.140 seconds of real time un time (0.508923 user, 0.004999 system)
  [ Ru6218 seconds of to al run time (5.092226 user, 0.053992 system) GC time. ]
  [ Run times con ist of 0.415 seconds GC time, and 4.732 second  non-GC time. ]
  100.12% CPU
  13,674,724,100 processor cycles
NIL,315,967,832 bytes consed
NIL                          enerations 10000 nil 10 1000 10000))
CL-USER>
1-U:**-  *slime-repl sbcl*   Bot (625,9)    (REPL Autodoc mate)----------
(do-generations generations &optional verbose (number-of-bags 10) (bags-of-size 100
```

Band distribution of numbers

0 < 60%X'es < 20%; 60% < 20%X'es < 80%; 80% < 20%X'es; 10000 generations

```
  747,961,880 bytes consed
NIL
CL-USER> (time (prt-spk::do-generations-with-band-data 10000 nil))
before: (7220246 7194660 7201368 7198508 7190395 7210115 7211425)(max-min-deviation: 29851)
after: (7203807 7203824 7203816 7203819 7203816 7203818 7203817)(max-min-deviation: 17)
Evaluation took:
  29.064 seconds of real time
  29.075579 seconds of total run time (28.389684 user, 0.685895 system)
  [ Run times consist of 2.969 seconds GC time, and 26.107 seconds non-GC time. ]
  100.04% CPU
  77,311,354,412 processor cycles
  7,488,275,552 bytes consed
NIL
CL-USER>
1-U:**-  *slime-repl sbcl*   Bot (541,9)    (REPL Autodoc mate)-------------------------------
```

Thank you

We are patch hungry*.
Please Contribute.
(its BSD 2 clause)

Thank you.

References:
http://test-load-balancer.github.com
* http://code.google.com/p/tlb/issues/list