

Title: Requirements/Design Document for Binary Regression Testing

Author: Judith Lebzelter

Date: October 25, 2004

Status: Pre-Draft

Overview:

The F/OSS Linux® stack has suffered from the testing of binary applications occurring in a way that is incompatible with the mainstream development processes. This is caused by the misalignment of the F/OSS and proprietary development cycles. With the former being measured in days and the latter in terms of calendar quarters.

This project intends to align Binary Regression Testing (BRT) with the Linux development cycle. This will require BRT to be done in such a manner that results are available within hours of a decision being made as to the F/OSS that needs to be tested.

In an attempt to bound the problem only a critical set of F/OSS packages composing the needed base of a deployed Linux solution will be tested. This limits the total package count to the low hundreds instead of the twelve to fourteen thousand F/OSS packages available on Linux.

There are several critical factors in this project. The first being ISV participation. The second being able to produce information that is useful to the F/OSS development community and finally being able to automate as much as possible of this cycle in order to meet the time constraints.

This document attempts to describe use cases, requirements and design for the BRT.

General Procedure:

- Narrow down the set of packages to a minimal installation. (Of course, this means resolving the dependencies and creating a working system for ISVs)
- Identify an application or application set that would work as the "Flight Simulator" of Linux and get it working on the minimal installation.
- Continually test the stack while updating the "critical components" in a very controlled way. Critical components will probably be things like the kernel, glibc, gcc, etc. We don't want to change multiple variables. Distributed testing may be needed to test when several critical components are changing. Success will be if defects that impact binary compatibility can be identified within xx days of the change.
- Periodically, test the application(s) with working set updates.

Basic Assumptions:

The purpose of the Binary Regression Test (BRT) project is to be able to execute regression test suites focused on specific proprietary binaries sooner in the release cycle than it is currently happening anywhere. Successes and failures will be mapped versus the predefined set of software packages upon which it runs. A subset of these packages will be monitored and the test will be executed whenever they have changes.

When failures do occur, information to be able to regenerate the system and regarding the cause of the failure will be captured. The source of the failure will need to be determined. This could be a problem in the application being tested, the test being run, or any of the underlying packages.

The major goal is to shorten the passage of time in the development cycle between when a code change happens and when it is noticed to have an adverse effect on the execution of the binary.

Make the base installation relevant to the distributions. If the distributions don't stay in sync with changes that impact binary compatibility, the effort is fruitless.

The success of this project will depend on getting people involved on a technical level outside of OSDL, preferably to help in the initial definition of the project and later to help in setting up SUT's, defining package sets and running particular tests. Getting technical involvement and possibly commitments should be incorporated into the initial sales pitch.

This project most closely resembles a multi-dimensional Tinderbox model where the trigger is based on one or more software packages. When a change occurs, the system gets upgraded software package(s) and a test or series of tests are run. The results are uploaded to the results database. The data is presented in a view similar to Tinderbox, over time showing passes and fails for a single host/test. However, there is a column for each package which is being tracked.

The data presentation will need to be more flexible than Tinderbox's: the user should be able to create views according to distribution, a particular binary or version of a binary, a library or version of a library, a particular SUT, or possibly others.

Monitored packages for each test will be defined according to the software being tested. Some will probably be common to all package sets, i. e. linux, gcc, glibc, binutils. Other libraries and packages may not.

Granularity for testing will need to be determined: is the trigger set according to releases or source control check-ins or something else. It is not certain that this can be completely automated when a project is extremely active more intelligent decisions about when to test might need to be made.

Beyond the simple pass/fail and the package set information, what should be retained? Possibly logs for the product installation, system logs, test outputs errors, etc.

The SUT should be designed to be able to run remotely and send results via either e-mail or SOAP/rpc. It should be relatively easy to set up, with a minimum of dependencies. It should run on various architectures and possibly distributions. To remove the issue of variation, we are going to have to go with a list of preregistered systems.

We need to be able to install the critical packages, and remove them. The SUT needs to be able to assess its own state: What critical packages are actually installed versus what is supposed to be installed.

The minimal installation is equivalent to a minimal distribution which we have thoroughly classified and documented. This is the initial set up for a test environment.

The main ISV binary under test has dependencies. These dependencies may also have dependencies. A user should be able to view all the dependencies in a format that makes the dependencies clear.

Use Cases:

Developer of a CP

- A CP developer makes changes. They check to see that all BRT's passed.
- After a release, a BRT is executed on the system with the latest updates. It fails either to build or execute. Someone (BRT Contact) must gather the information concerning the failure and get it in a coherent form to make a bug report on the CP project. The CP developer treat the bug as usual with the BRT contact as the primary contact.
- Before an official release, the developer creates a pre-release which is run through BRT. The developer checks the results for success before making the official release.

Developer (Open Source) of Application that depends on a CP or ISV Binary

- A developer needs functionality that exists in an updated library. They want to be certain the updates will not break a key application.
- They check the library in BRT for other dependencies.
- They check for test successes including the new version of the library.

End administrator of a particular application

- A n administrator needs to upgrade the system but does not want to break their application. They check on issues with updated software in BRT for the release version they are interested in. If it is acceptable in combination with their system then they update their system. If it is not, then they check what is a successful combination and do further upgrades.

- Do the "could this work" check

Distribution

- Check for package versions for critical ISV's
- Patch check - for known updates that can cause application failures.

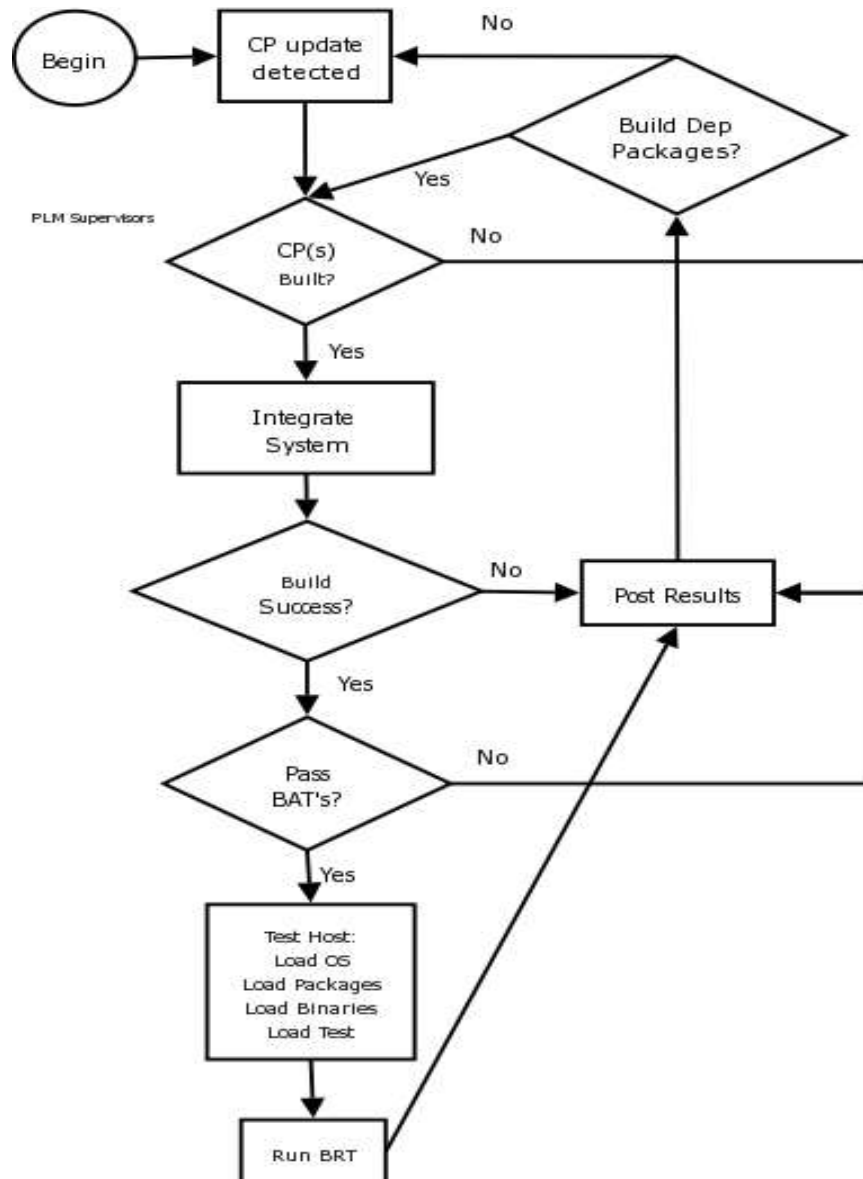
ISV Developer or Integrator

- Check for patch/update issues. i.e. Security patch comes out. Does it break anything? They receive notification if it is a problem. They check the results of the proprietary test and the tool kit which gathers information on the failed system. They attempt to narrow down where the problem is occurring: the test, the binary, or the patch. Once the problem is narrowed down the appropriate channel is notified in the best manner to fix it.
- Distribution pre-check. Are there known issues with a packaged distribution?

Process Flow:

Figure 1 (below) shows the action flow in the BRT cycle. A change in a CP triggers the testing of that CP. If that test succeeds, the integrated host is built. If that succeeds then Basic Acceptance Test are run. If those pass, the the Binary Regressions tests are run. Failure at any point means results for that Bill of Materials (BOM) are posted. A decision may then be made whether to re-build more dependencies and run again.

Figure 1: Process flow for BRT Cycle.



Architecture:

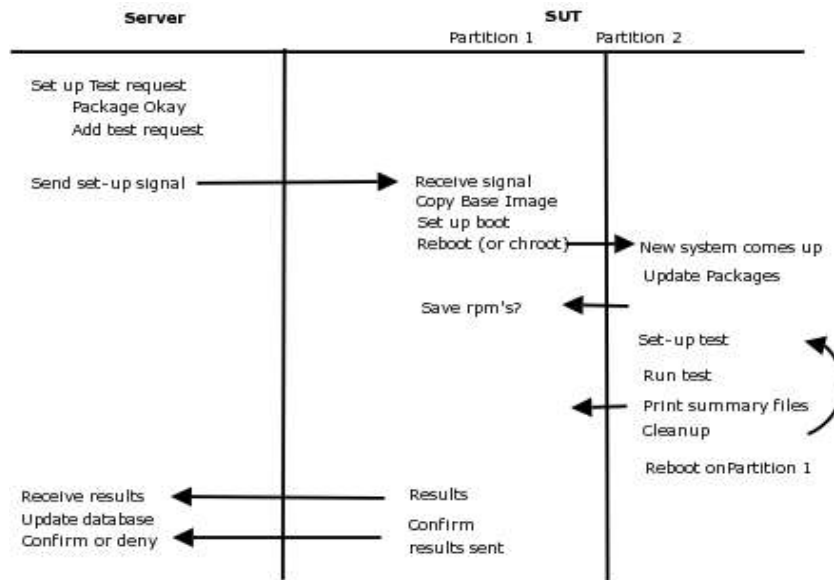
This seems similar to a continuous integration/testing sort of problem to me. However, instead of one repository, the binary under test and all its dependencies are monitored for change.

Client Side - To remove the issue of variation, we are going to have to go with a list of preregistered, well-defined systems, to be a registered system id and we will need a test system table.

1. System information and test results capture must be automated and then submission to the central data collection point.
2. Get information for system configuration for test.

- Set up packages which are updated
- Rebuild entire system.
- The figure below show the SUT as a dual boot system. The 'manager' receiving signals and setting up the other partition for test. The 'test' side sets up and executes tests. This could be a 'chroot' rather than a reboot.

Figure 3: Server/SUT interactions



This is a diagram of the server and test system interaction. The SUT has a 'manager' boot partition and a 'test' partition. Both of these systems should be able to access the data for update or upload. We want to avoid the SUT polling, so we will need a system to send messages to the SUT, as well as the RPC to update state and results.

Server side

1. Receive information from SUT's, either via e-mail or call to SOAP server. There will need to be some kind of security in the information transfer.
2. Database to store results
3. Where do we store captured logs?
4. Presentation of results to the audience.
 1. Search results in database
 2. Web presentation, like tinderbox with a column for each product/dependency.
 3.
 1. Pass out information to client on system to update.
 2. Have knowledge about when updates are necessary

List of Servers Defined in E-mail from Tim:

1. Storage: Unit server for storage attach. If more storage is needed the

- additional servers will be brought on-line.
2. Tinderbox: Compile machine – multiple compile machines. Code loaded from storage units for compiles. Machines added until build cycle meets users needs. (renamed to 'Build Farm1')
 3. Build: Integration machine final integration and test version builds are done here. (renamed to ' Intgration')
 4. Web: Machine to provide web services
 5. DB: Machine that houses test database.
 6. Test 1: Target test machine

Figure 2: Initial architecture diagram

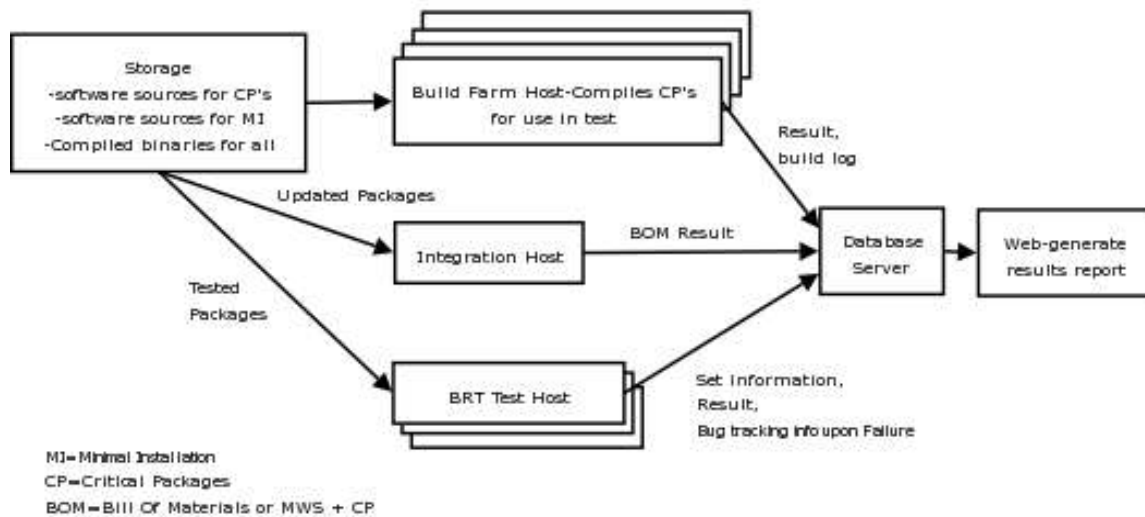


Figure 2 shows the initial impression of for the architecture as envisioned from Tim's presentation and from e-mail on the Binary SIG. This is not finalized and not fully defined as of yet. However, here is the process flow for a successful test cycle:

1. This starts with a fully defined Minimal Installation (MI) and one or more defined Critical Packages (CP).
2. We are monitoring the CP's and we notice that one has a new release.
3. A trigger starts a compile in the Build Farm of that CP, and perhaps a compile of any package in the whole BOM which depends on that component. In this case the build state is as important as the version of the package.
4. Build Farm sends results to the BRT application which acts as a gateway for the next step. (Stop here if it fails)
5. A trigger initiates installation and basic acceptance tests (BAT) on the host 'Integration'. It gets packages from 'Storage', installs them and runs BAT's.
6. BAT's results are sent to Database.
7. If BAT's are successful, a trigger sets of the BRT's on 'Test'. 'Test' runs a

machine check, gets any package upgrades from 'Storage', installs sets up test and executes it.

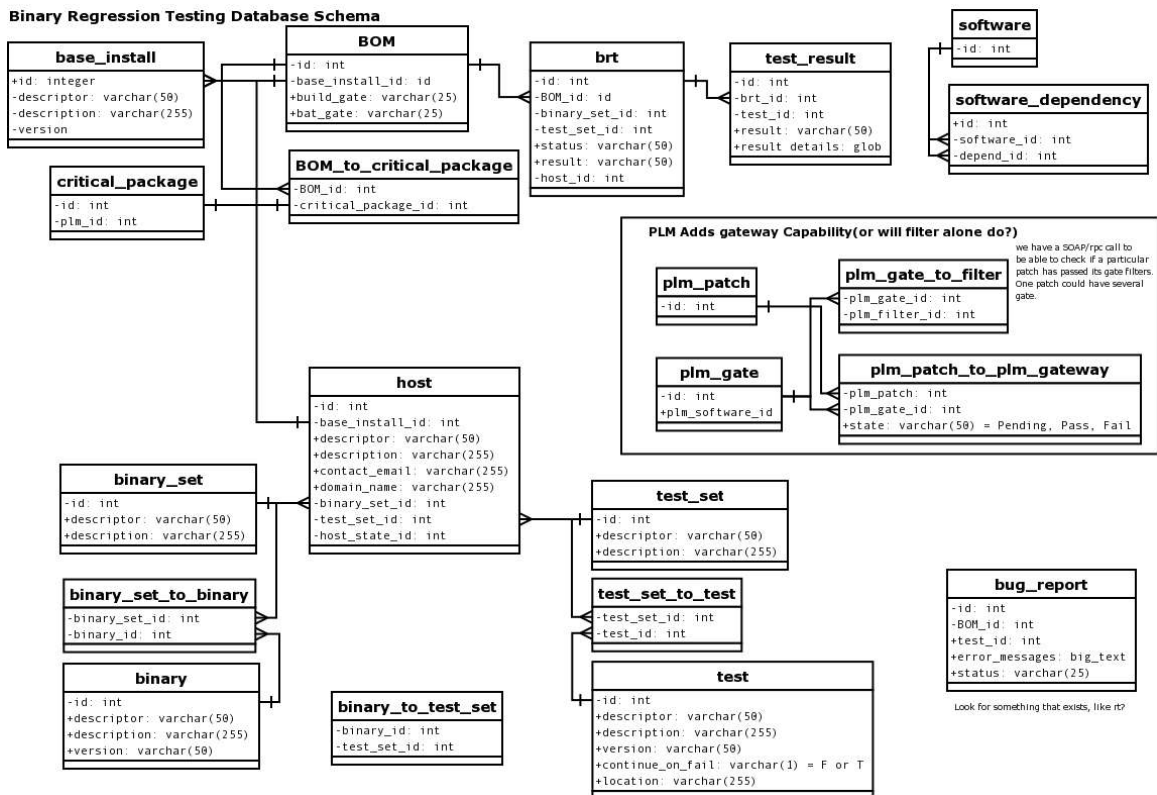
8. Results are sent back to 'Database'. If there is a 'pass', the cycle ends here.
9. If there is a 'fail', enough information to regenerate the system and point to the problem must be saved. Defect analysis, bug reporting and follow-up must occur.

If we are not using the build farm to generate binaries for download, then it seems that the build farm is actually a lot like plm supervisors, but that we are actually using the filter results to gate the next step.

If we want to generate binaries for download, then we get limited to architecture,

System and state information will lbe stored in a database, probably MySQL to match the other projects. The beginnings are layed out in figure 3 below and will be based as much as possible on reuse of the current STP and PLM.

Figure 3: Database Schema



This project can be divided into logical development portions:

- Research and chose Minimal Installation (MI), and methods of tracking and installation.

- Testing Framework
 - Define information for result uploads and format to send
 - Define method of communication between SUT and server.
 - Client actions, must be automated
 - Installation (Optional?)
 - Machine check prior to system set-up
 - System upgrade
 - Load binaries, tests, set-up
 - Execution of Test(s)
 - Upload Results, formats and storage
 - Server Actions
 - Interface with database
 - Assign tests
 - Initiate client action, or does client poll?
 - Receive test results, parse into database or files.
 - Administrative
 - Registered/Decommission SUT's
 - Add Tests/Test Sets
 - Register Binaries/Binary Sets
 - Track 'Critical Package' sets and their dependencies
 - Track distributions-including 'Minimal Installation'

- Results Presentation-should this be web or mining tool.

- For each binary test a development/set up sub-project must happen.
 - Define 'Critical Package' set
 - Set up each CP for build/installation automation
 - Set up Application for 'build/install' automation
 - Set up test automation
 - System clean-up after run

- Defect Analysis and bug tracking System.

- Outreach to ISV's and Distributions.

- Outreach to Developers of 'Critical Packages'

Major Questions and Points:

The following is a list of questions or issues that need to be answered/addressed before the design can be considered complete.

- Are there any requirements that force us to rebuild the system from scratch for each test? These are purely regression tests, so prior system history should be less important. There can also be a clean-up script after testing. It would be more efficient to update (or regress) only the appropriate critical packages.

- How would we set up rules for requesting combinations of packages? And for rolling back the version on test failures. (If we do not re-build each time) Or maybe uninstallation?
- What sorts of triggers would be most appropriate for the packages? Are they all the same or do they need to be defined separately. What actual level of granularity do we want, released packages or source code check-ins?
- Can we build this modularly, in stages to be able to (usefully) use portions before the whole system is together? Or provide tool that might be useful to other projects? If so define the major sections and prioritize.
- For each test we will need to know what we can publish and what we can't publish. Also, how to get a simple pass or fail result from rather complicated tests or test suites.
- It is desirable to be able to run remote test hosts. This would allow ISV's, Distributions or perhaps users of the binary to be tested to provide and maintain test systems. The BRT would be much more scalable and could collect data on more types systems, assuming there is interest. Also, tests would be run where the most domain knowledge for the binary exists. A reason not to design this way would be that some essential functionality could only be provided locally, forcing us to have all test machines be located at OSDL, which I do not believe to be the case.
 The SUT's stability is a major issue especially if they are remote. Having a stable installation on one mount point which is the control system and will initiate the test system install on another partition would be great.
 Also a 'does it boot?' BAT would be nice, in an environment where we can time it out and restart it without manual intervention.
- How do we resolve issues when a test or build fails? Do we have enough internal engineers to submit issues to projects and follow up on them, possibly maintain relationships with important projects? What procedure do we follow to do this. How do we really short circuit the development cycle from release to bug report to bug fix the most effectively and efficient with our limited resources?
- How will we define the 'Minimal Installation' of packages? This is sort of like a minimal, no frills, distribution. Is there Should we limit tests to be running on the Minimal Installation (MI) or should that be an option as long as the host information is recorded as part of the test results along with the CP conditions?
- How do we define the 'Critical Packages'? We need to write guidelines and procedures for how to do this.
 - Identify Direct Dependencies
 - Libraries-use 'ldd'

- Tools
 - Applications
- Indirect dependencies-repeat process for any dependencies found
- Some applications distribute with their own versions of libraries. We must be certain that we are testing what we think we are testing.
- Source trees and patches will need to be accessible from the internet in order to allow external SUT's.
- If we want to use PLM for the source access, it would be good to add
 - a display table for each filter type that shows the filter results summary in a series for all patches ordered from the most recent each.
 - We probably want two basic filters for each, an 'applies' filter, and a 'basic compile' filter. We should consider this for any source that needs to be built.
 - We need to be able to search by any to see all recent patches.
 - Do we want to use its installation features, or will LFS or Gentoo be better?
- STP has the server side set up the test and the re-image and then reboot the SUT. The SUT PXE boots with the chosen image. The BRT architecture needs differ from STP in several ways since we want the client to be remote. Currently, a dual boot option is being considered. If the remote server is partitioned and has a dual-boot image. One receives the 'setup' command and creates the image, then reboots on that image. The second does any more setup, tests the state and runs any tests. Either system could do result upload and clean-up; however the first image could also do recovery if the test system becomes inoperable.
- Communications: We could have daemons on both ends in a client/server model: the client would wait indefinitely for update information and the do check set-up and execute, upload and cleanup. Another possibility is the SUT could have a client polling the database through the SOAP/rpc server daemon or cgi. The cgi option is less favored for a reliable application. We could have a set up like current STP, host-state stored in database or have the server keep track of state information. What states should we use? Since there will be little required interaction between the SUT and the Server, fewer states will be necessary. We may want to still have the state timers functionality, but what the server can do on time out is limited: ask the SUT to end the test, upload and clean up, record what it was doing on timeout, send mail to the admin to indicate a timeout/failure.
- Builds and installations for the packages can be pretty tricky. We need to check the dependencies and automate the installs, especially for difficult packages like glibc.
 - 'Portage' as a build tool looks mature and tracks dependencies in files. The

distribution 'Gentoo' is built on portage , but it indicates it is a 'very powerful advanced package management system'. It already has all the dependencies for each build and between the packages for more than 8000 ebuild recipes. (so we will need to pare that down, a lot, look at LFS) If we want to have this information in the database, we should write a script to parse these files. We can also write a script to edit the files in order to update the system. This systems pulls tar balls and can do RPMs. I am not so sure about cvs or bk. We may need a work around.

- Gentoo as an available installation to the test hosts could be stored on 'storage' after stage 3 has been built on the integration host or hosts. If we want more than one architecture we need a host for each or a very fancy build-the-cross-compiler then build the gentoo set-up.