CSE 503 HW1: Development Difficulties

Everett Maus (evmaus)

April 1, 2017

1 Distributed Systems

Throwing more processors at a problem to solve the limitations of running something on a single core or single computer is unlikely to go out of style anytime soon–more and more systems require the availability and scale that can only be achieved by running something on many computers and many cores. However, ensuring a distributed system is relatively bug free is incredibly difficult at every stage of development. During the design of system, many initial specifications of a distributed algorithm can be incorrect in subtle, hard to detect ways. As the system is written, the programmer needs to be able to reason about sets of states the system might be in at any point of execution. Lastly, once a system is in testing/production, there are often rare edge cases (e.x. nodes disappearing, low memory conditions leading to the host OS killing a process) that are difficult to simulate or which only occur rarely.

One option for addressing these difficulties is to prove the algorithm and implementation correct with respect to whatever properties you care about. However, verified programming is often more time intensive than traditional implementations, and requires using specifically designed programming languages such as coq, which may not have the same library functionality as a more mainstream language–the UW Verdi implementation of Raft required proofs between 5 and 10 times as long as the actual implementation. Verified programming also can introduce higher maintenance costs, as a small change may require the entire system to be re-proven.

Another approach for ensuring systems are reliable is to inject what would otherwise be rare faults (such as machines crashing, network unavailability or high latency) into a full deployment of the system–an implementation of this model is Netflix's "Chaos Monkey" system. However, this inherently cannot catch all possible errors, and has the limitations that testing in a production system put on the developer–it's harder to debug, harder to reason about the past state of the system when a failure occurred, etc. This also is unlikely to hit any extreme edge cases which could require a specific set of user actions to trigger (a category which describes many security impacting bugs).

A middle road is to use a specification language like TLA+ to specify and prove the algorithm correct, but use an unproven implementation. Amazon reportedly used this approach successfully with systems like DynamoDB. This can catch any algorithmic problems, but as the specification isn't linked to the implementation in any way, you still have the challenges of ensuring that the implementation is correct.

Something that "crosses the gap" between a formal specification of a distributed system and the actual implementation, without requiring a fully verified implementation could help address some of the limitations with the current options. One option would be lifting a specification from the implementation and ensuring it meets the properties you care about. Another possibility would be to create something that turns a specification into partial or complete code–a "specification compiler" or a set of libraries in a standard language which are linked a set of primitives you can use to specify a protocol/algorithm. An example of a very limited application of the second approach is visible in Google's "Paxos Made Live" paper, where they created a DSL for specifying the Paxos algorithm, which compiled down to C++, in order to separate the algorithm from implementation/communication details.

2 Refactoring for Testing

A common problem in system maintenance is taking an existing system and porting it to a new context or extending it when the original developers are no longer available to answer questions about it. A com-

prehensive unit test suite can quickly answer questions about if those changes are likely to break existing functionality.

However, if a system hasn't been designed with testing in mind it's difficult to add tests to it after it's been completely written without a major refactoring effort. Consider a hypothetical system that has several dependencies on external systems–if those were not initially abstracted away or mocked, it's often difficult to refactor the code to abstract them or mock them in a way that allows someone to write unit tests cleanly.

Automated refactorings can help with this–automatically replacing class usage with interfaces and automatically creating mock classes for unit testing. However, these often have some limitations–for instance, if a system is interfacing directly with a SQL database and expects a series of queries to first store data in a particular format and later retrieve it, it's unlikely that an automatic mocking system will accurately model that behavior. It's also difficult to automatically mock remote services–often both of these scenarios require a developer to manually create mock classes which simulate just the responses they expect, or to replace the usage of explicit calls to a database with an Object-Relational Model or similar framework.

A combination of approaches can help address some of these challenges. One option for very standard infrastructure services, such as a database, would be to create local test implementations that do a reasonable job of simulating the actual service–e.x. using a file based database like sqlite or an in-memory database as opposed to a full database server. This has the downside of being a simulation, though–any dependency on specific services may also need to be simulated, which could run into implementation issues. For example, a service dependency on Microsoft's Sql Server Integration Services for moving data from one SQL database to another would need to be simulated–after a certain point you're running the server you wanted to avoid creating.

Another approach would be to run a potential test against a real endpoint and then cache the results for specific test requests in a mock class. This puts a number of limitations on your tests (they must be truly deterministic–no randomizing elements to better simulate real environments) and risks drifting from the actual service's behavior (although that winds up being a problem regardless of your tests with an external dependency, but in this case it may get caught later on).

3 Execution Privileges

As a general principle, executing code with the smallest privileges possible is a security best-practice. However, this is often hard to implement correctly, even just for a single process running on a single machine. The problem becomes much harder if you consider a service running in a network with some sort of managed/centralized identity model, which needs to be able to authenticate to other services.

One option that many services take is to take a greedy approach, and simply ask for more permissions than a service might need, in hopes of avoiding crashes or reduced functionality caused by not having sufficient permissions. However, although this is effective it obviously violates the general principle we wish to follow.

Another standard approach for addressing this are to start testing with the lowest possible permissions, and slowly elevate your application's permissions as you hit operating system or service errors. However, this requires careful debugging, a deep knowledge of both your system and of the permission systems your system lives in, and is quite time consuming. This also runs into issues with rare execution paths–if you didn't exercise a particular piece of functionality and it requires permissions you didn't grant the application, you're might run into issues down the line. Furthermore, since this is time consuming and difficult, it's often done once and then further permissions are only added when you find you need them, leading to a slowly growing list of permissions that an application has (which is rarely revisited, if ever, to remove permissions that are no longer required).

A third standard approach is to list all permissions you think your system needs, along with a justification of why your system requires that permission. This is manual and error prone, however, and often simply provides a better starting point for the second approach. It also has the challenges of keeping it up to date that the second option has.

One approach to solving this could be to offer a way to annotate the permissions that a particular piece of functionality needs to execute–then the total permissions required by your application are the permissions required by each piece of functionality. This could also allow you to zero in on particular pieces of functionality in an effort to reduce the overall permissions required. However, since permission systems

are heavily integrated into the operating system or network and are by design very granular, this sort of annotation may be quite difficult. For example, most programming languages use the same functions for writing to a temporary folder and a more privileged folder (e.x. /etc on linux systems, or 'C:\Program Files\' on Windows systems), so finding the appropriate way of representing the permissions that a function requires may be difficult to handle at design/compile time.

Another possibility would be to offer a dynamic "testing" identity, which gives the user all possible permissions, but logs all the ones which are actually required as a program executes. This primarily offers a faster way to approach and iterate on the second approach. However, creating an identity like that on a machine (or worse, on a network) opens the possibility that someone could use that testing identity as a superuser and get access to resources they shouldn't be able to access. Controlling access to the test system then becomes a challenge. This also, similarly to the second approach, has the downsides of rare/untested code paths going unexamined.