1 Describe the architecture of a possible automation testing system

There are different aspects of software that are typically subjected to testing. The categories are roughly as follows:

- Functional testing. This ensures that the product matches specification.
- Performance testing. This ensures that the product doesn't exhibit pathological behavior when run over long periods of time. In addition to above, such tests usually try to establish that the product adequately responds to change in pressure from resources or input.
- Benchmarking. This tries to establish how well the product can utilize available resources.

Not only all of the above require different approaches, these aspects have further sub-division, that, in practice, often leads to different, unconnected, implementations. For example, functional testing is often envisioned as a hierarchy of scope of a single test:

- Unit testing. This deals with the essential and indivisible building blocks of the software being tested. Such tests are designed to be executed by software developers w/o the aid from Q/A as part of their development cycle.
- Acceptance testing. This is usually a small group of tests that is specifically designed to cover as many features of the product in as little operations as possible. The goal of such testing is to serve as a gatekeeper for Q/A, as often times sending a product to Q/A is expensive, while "dead on delivery" software will produce very little value proportionate to the effort spent. This sort of testing is often dubbed as "sanity" testing. Such tests are designed to run whenever a features is merged into a common code repository.
- Nightly (alternatively: periodical testing). This testing is designed to exhaustively search for defects in software, a separate part of periodic testing is regression testing, which requires its own treatment. This testing is the bread-and-butter of Q/A, as this is where most non-trivial bugs are found. Another aspect of periodic testing is that it usually tries to test the integration between multiple components,

where end-to-end testing is a special case of trying to engineer complete usage scenarios. Yet another aspect of periodic testing is "error injection", i.e. tests that examine system's behavior in situations where hardware components fail, or software components outside the system fail / provide faulty input.

In the light of the above, it's very optimistic to hope for *a* testing system. There aren't real testing systems that attempt to cover all the aspects of software testing, and it's hard to imagine what such system would be like, should anyone attempt making one.

This is why I will interpret this task as asking for a design for some aspect of testing. I will choose end-to-end testing, as it is my strong conviction that of all kinds of functional testing, end-to-end provides strongest guarantees (e.g. the product is shown to work at least some times), while other aspects cannot promise even as much.

1.1 End-to-end testing for BitTorrent client

1.1.1 Which frameworks would you use to write the automation?

I'm not aware of existing testing frameworks that go beyond unit-testing. Unit testing received a lot of attention from the programming community, while other aspects of testing have very sketchy coverage. In my experience, all systems that purported to serve this goal were in-house development, that never transferred from product to product. Perhaps, a general system could be created for such tests, but this would be clearly beyond the scope of this exercise.

1.1.2 Which tools would you use to run the BitTorrent that you're testing?

I assume that I'm testing a BitTorrent client, and that the functionality of, for example, trackers is out of scope for this exercise.

The way I'd approach this is by adopting and extending `etorrent` client `https://github.com/jlouis/etorrent` . Erlang is chosen for the provisions the language runtime makes for distributed computation, the embedded relational database, minimal requirements for installation and a good track record of creating testing frameworks (eg. QuickCheck).

Using `Erlang's` OTP would allow me to abstract the deployment process from the actual hardware (or virtual hardware) on which the tests should run, I would also be able to use presets to configure networks, as instead of creating custom networking layer for tests, in essence, I'd be configuring the networking layer for OTP.

I would also need access to the version control tool used by the development. I'll assume `Git` is used.

I would also need access to bug tracker software used in the project. I'll assume `JIRA` is used.

I would also need a configuration tool to set up multi-host environments. I'll assume `Terraform` is used.

Depending on company's financial situation (in general, public cloud services aren't cheap, but it's possible to get exclusive contracts that mitigate the costs), I might thus use vendor-specific software to connect to cloud services, or use on-premises lab equipment if such is available.

To speed up development process, and since hardware emulation is hardly essential to BitTorrent client, I'd use some container technology to simulate multi-host networks on single physical host. `Docker` seems like a popular choice here.

Finally, I'd need a test runner to deal with the test maintenance issues, s.a. sending alerts, keeping logs and artifacts, managing test runs and providing interface for developers to execute tests and evaluate test results. There are plenty of projects that try to fill this niche. Unfortunately, none are any good. `Jenkins`, however, is a popular choice.

1.1.3 When should the automation project run?

If end-to-end tests take less than 24 hours, then they should run daily, if they take less than two days, then bi-daily, and so on. This is, of course, subject to financial constraints.

1.1.4 Create a simple sketch of the framework, naming all its parts (binaries/machines)

Deployer The test module responsible for deploying SUT (system under test) and the test itself. This is the module that contains `Terraform` scripts and `Docker` images necessary to create test assets.

Runner The test module responsible for execution of the test plan. It should receive input from `Deployer` describing the SUT, and organize tests in such a way that they use given system resources.

Analyzer The test module responsible for collection of test results, their refinement and shipment to persistent storage. This module is also responsible for providing initial triage and RCA (root cause analysis), as well as alerts.

Interface The test module responsible for implementing the client side of the interface with the SUT. This module provides functions and other definitions used across different tests to perform actions on the system. This module also contains the model of the system as well as the code to assess that the system is in acceptable state.

Harness The test module to deal with the administrative side of tests: the connection with `JIRA` and `Git`. This module should be able to, given input from developer, select the tests relevant to the feature being developed, filter the tests that are known to fail due to known bugs, update reporting systems, initiate code reviews, or prevent merges. This module is also responsible for providing testing primitives s.a. `suite`, `scenario`, `step`.

Individual Test Modules For example, `E2E` module would be responsible for describing scenarios developed by testers to that end. Testers would be thus responsible to use the functions and other definitions defined in `Harness` module to structure their tests and functions and definitions from `Interface` module to interface with the SUT.

2 Create a Software Test Description (STD) Document, sorting tests by section/scenario

While story / scenario approach is popular in the industry today, I don't believe in it. My argument is that such approach provides indeterminate coverage and creates a lot of busy work for Q/A. Not only busy work is a problem in itself, it also creates an environment that welcomes people of low aptitude. This creates a vicious circle, whereby testers are treated as less capable programmers, while programmers, on the other hand, will never actively participate in testing. This results in relative uselessness of tests, where programmers, knowing the tests to be mostly useless will implement ad hoc, poorly supported and poorly documented framework to cover their individual needs wrt' testing.

I'm a proponent and supporter of model-based testing, where the objective of the tester is to design a model of the SUT, while describing as precisely as possible the properties of the SUT on one hand, while, on the other hand, the testers need to implement the runner that systematically and automatically verifies the properties of the system. So-called "fuzzy" testing or "property-based" testing are thus steps in the right direction.

However, I also understand that it's hard to convince people to do things in an unusual way. It is also hard to find Q/A professionals skilled in this approach, this is why I'm going to describe the testing plan in a more traditional way to the best of my ability.

2.0.1 Make sure you mention ALL the necessary sections

As I already mentioned before, story / scenario approach isn't the one that can provide exhaustive testing. Not only that, it's impossible to even tell what fraction of functionality is being covered.

2.0.2 You're NOT required to write the steps of each scenario, but make sure you explain the major scenarios in each section.

I will use Gherkin to give examples of possible scenarios. Gherkin is a language invented by Ruby TDD televangelists. I'm not affiliated with them, but it might be a convenient common ground.

```
Feature: Publish To Tracker

  Scenario: Publish To Multiple Trackers
    Given I deploy the system
      | image       | role    | id | location  | platform |
      | tracker.ova | tracker | t1 | us-east-2 | aws      |
      | tracker.ova | tracker | t2 | us-west-2 | aws      |
      | client.ova  | client  | c1 | us-west-1 | aws      |
      | client.ova  | client  | c2 | us-east-1 | aws      |
    When client "c1" publishes to trackers "t1,t2" file "test"
    Then client "c2" merges download from trackers "t1,t2" of file "test"

  Scenario: Handle Network Partition
    Given I deploy the system
      | image       | role    | id | location  | platform |
      | tracker.ova | tracker | t1 | us-east-2 | aws      |
      | client.ova  | client  | c1 | us-west-1 | aws      |
      | client.ova  | client  | c2 | us-east-1 | aws      |
    When client "c1" publishes to trackers "t1" file "test"
    And client "c2" download file "test"
    And client "c1" starts downloading file "test"
    And client "c2" is stopped
    Then client "c1" has peer list
      |  id |
```

Obviously, there are countless many possible scenarios like the ones shown in example above. For instance, you can indefinitely expand the last scenario by bringing `c2` online and sending it back offline, thus producing, while not very meaningful, definitely different tests.

Similarly, you can, for example, add more trackers to see if downloads are merged for more and more trackers. Not very useful, but, definitely different.

3 Without ever touching the BitTorent system, please describe 2-3 potential bugs that are likely to be found the system?

I'm not sure what does "ever touching" mean in this context. Also, the sentence is missing a preposition before "system". Also, I belive that "BitTorent" is a typo, should've been "BitTorrent". I'll assume that "without ever touching" means that I should pretend I know nothing about BitTorrent (I know very little indeed, so that's not hard). I'll assume that the missing preposition is "in", i.e. bugs are found in the system.

There are indeed many problems virtually any program may encounter, which are unrelated to the nature of BitTorrent protocol / system. For instance, a program will usually take some arguments when started, on UNIX systems arguments come from two sources: supplied through environment, and supplied through command line. Some trivial failures will include handling Unicode or non-Unicode input.

Another trivial and general thing to check is whether a program responds well to running multiple copies of it. I.e. try starting another copy of the program once one copy is executing. Some system resources are unique, sockets, locks, CPU cores etc. When the program isn't designed to deal with other copies of itself it may crash or stall instead of informing the user the action cannot be performed.

Yet another trivial problem is the one of permissions. Some programs may assume free access to system utilities which might not be available due to user policies on a particular system. Thus it's useful to try running the program as administrator, or as a unprivileged user.

It is hard, however, to find useful tests that are general enough not to require any specific knowledge of the SUT.