ToDD Documentation

Release 0.1.0

Matt Oswalt

Oct 24, 2019

Contents

CHAPTER 1

Introduction

Welcome to the documentation for ToDD! ToDD is an extensible framework for providing natively distributed network testing on demand. ToDD is an acronym that stands for "Testing on Demand: Distributed!".

Traditionally, the tooling used by network engineers to confirm continued network operation after any kind of change to the network is fairly limited. After a change, a network engineer may run "ping" or "traceroute" from their machine, or perhaps call some application owners to ensure that their apps are still working. Unfortunately, there is a significant difference in network activity between a 3AM change window and peak user activity during the day.

ToDD addresses gaps in today's testing software in three ways:

- Enables real-world traffic distribution for tests using simple grouping primitives
- Provides a open and extensible mechanism for defining tests
- Exposes testing data in an open way, easily allowing for 3rd party software to analyze and visualize

ToDD is a framework through which you can deploy simple test-oriented applications in a distributed manner. With ToDD, you distribute agents around your infrastructure in any place where you feel additional "testing power" is warranted. Then, these agents can be leveraged to mimic real-world network utilization by actually running those same applications at a large scale.

Here are some key features provided by ToDD:

- **Highly Extensible** - ToDD uses a generic interface called testlets for running applications, so that users can easily augment ToDD to support a new application.
- **Post-Test Analytics** - ToDD integrates with time-series databases, such as influxdb. With this, engineers can schedule ToDD test runs to occur periodically, and observe the testrun metrics changing over time.
- **Grouping** - ToDD performs testruns from groups of agents, instead of one specific agent. The user will provide a set of rules that place a given agent into a group (such as hostname, or ip subnet), and ToDD will instruct all agents in that group to perform the test. This means that the power of a test can be increased by simply spinning up additional agents in that group.
- **Diverse Target Types** - Test runs can be configured to target a list of "dumb" targets (targets that are not running a ToDD agent), or a ToDD group. This is useful for certain applications where you need to be able to set up both ends of a test (i.e. setting up a webserver and then testing against it with curl, or setting up an iperf client/server combo)

CHAPTER 2

ToDD Concepts

Before exploring the details of how ToDD actually works, it's important to first spend some time understanding a few fundamental concepts.

2.1 Groups

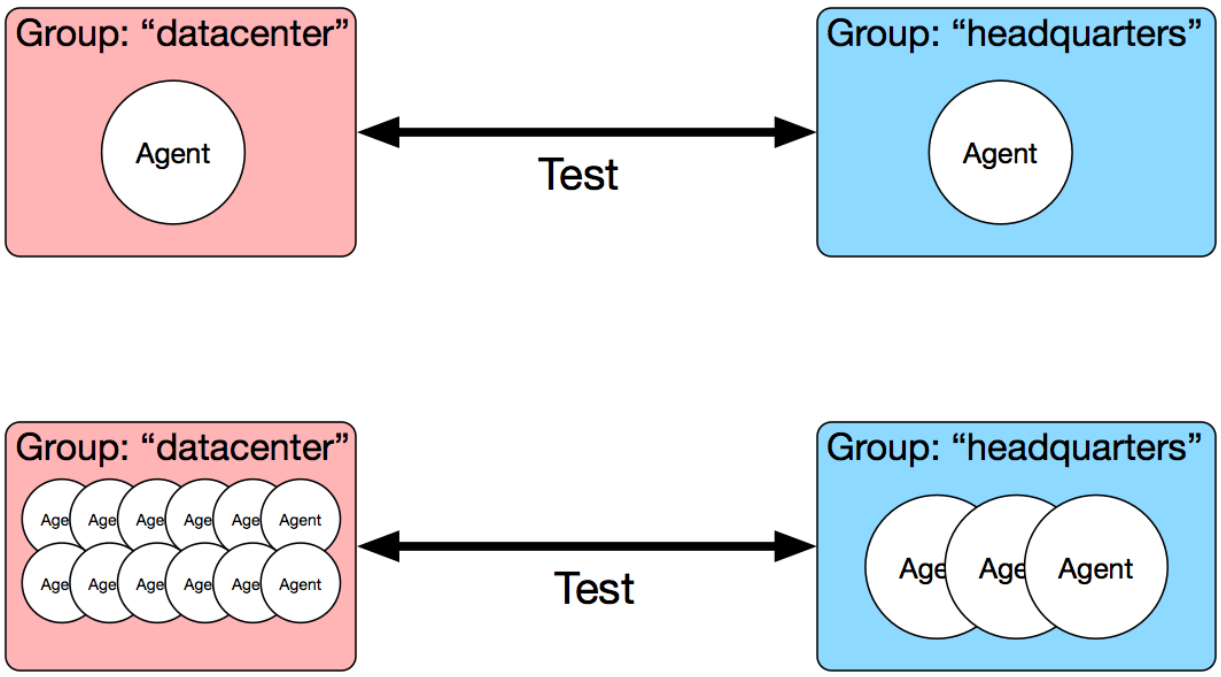
In order to solve the problems that ToDD was created to solve, it was necessary to consider scale from the beginning. In ToDD, testing is performed much more like "cattle" than "pets". As a result, tests are not performed with discrete endpoints, such as individual ToDD agents, but rather with "groups".

Note: At its lowest level, testing in ToDD is done either from a group of agents, or between groups of agents. It does not deal with individual agents. This is a very important concept to remember, as all other concepts build on this.

You should still consider a "group" as a singular logical endpoint for a test, even though there may be multiple actual endpoints within that group.

Groups can contain as few as one single ToDD agent, or as many as tens or even hundreds. The idea is to detach the concept of testing from discrete nodes or IP addresses, and provide a grouping mechanism that allows the ToDD user to adjust the scale or power of a test without fundamentally changing its parameters.

This concept shows its true power when you compare two instances of the same test being run. In the diagram below, we have two instances of the same test. In the first instance, only one agent is present in each group. In the second, several additional agents are present in each group:



Note that both the group configuration, as well as the configuration of the test itself, are identical between these two test instances. This is because the mechanism by which you describe groups in ToDD allows you to anticipate different “test topologies” ahead of time. Tests are performed between groups, regardless of what the group happened to contain at test time.

For instance, the group “datacenter” might be configured to include all agents that belong to a certain subnet. Once this configuration is in place, scaling this group to include additional agents is a simple matter of spinning up additional agents that belong to that subnet.

Note: See the “Group” section of the Objects Documentation for more info on what parameters can be used to assign agents to groups.

To summarize, groups can be best thought of as a scaling mechanism. ToDD assumes you have other tools, such as Kubernetes, for automating the creation of endpoints, and provides a way for you to describe logical groups for agents that are added to the system, based on their attributes.

2.2 Testruns

The “testrun” is the most atomic unit of testing in ToDD. It is the abstraction upon which all other forms of test automation is built.

In the previous section, we discussed how testing is done using ToDD Groups, and not with discrete agents. The testrun is where this idea manifests itself, as testruns are a logical point-to-point concept. They’re always taking place either between a group of ToDD agents and a group of “dumb” endpoints, or between two ToDD groups. While it’s true that there may be (and probably should be) many actual, addressable endpoints taking place in a test, the testrun is still describing a point-to-point relationship.

The testrun configuration is designed to answer three questions:

- What are the two ends of the test?

- What application am I testing with?
- What kind of parameters does this test need to be successful?

Note: See the "Testrun" section of the Objects Documentation for more information on the specific syntax used to describe a testrun.

Again, the Testrun is the most atomic unit of testing in ToDD. As a result, there are a few things that a testrun does **not** do:

- **Testing beyond more than two endpoints** - testruns are group to group only
- **Aggregate metrics** - all metrics are reported individually, per-agent.
- **Automatically run themselves** - testruns are either started manually via the CLI, or programmatically.

Testruns intentionally do not do these things because they're leaving room for future, higher-level abstractions to do them.

Note: The concepts that will be involved with doing these things are still a work-in-progress, and feedback on how these things should be done is welcome. In the meantime, ToDD tries to be as open as possible, so that you can solve these things for yourself until they're done in ToDD.

The ToDD Server will distribute testrun instructions to the agents in two phases:

- Install - run the referenced testlet in check mode , then record all of the parameters for the intended test in the agent's cache
- Execute - run the installed testrun instruction

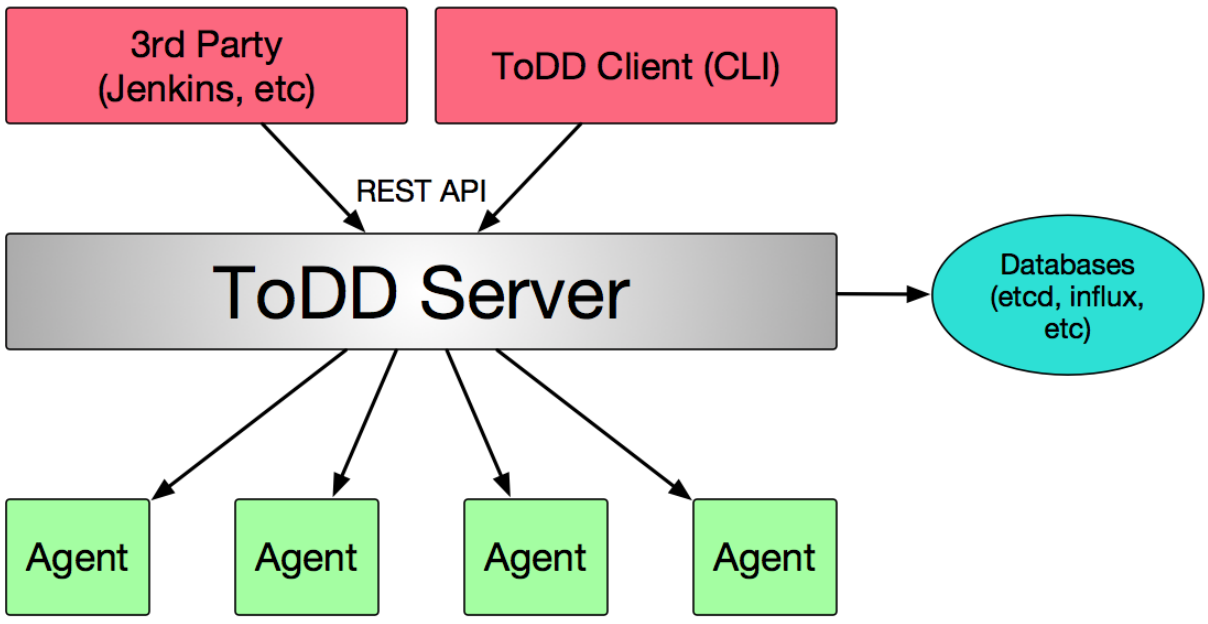
CHAPTER 3

High-Level Design

The High-Level Design of ToDD is fairly simple. ToDD is composed of three components:

- Server
- Agent
- Client

A general idea of how these components is depicted below:



Some notes about this:

- All database integrations are at the server level - no agent communicates with database

- Agents **do not** communicate directly with server. This is done through some kind of message queue (i.e. RabbitMQ) using ToDD's "comms" abstraction.
- Server has a REST API built-in. No other software needed (see section "ToDD Server" for more)

The following sections elaborate on each component in greater detail.

3.1 ToDD Server

The ToDD Server has a few particular noteworthy attributes:

- Orchestrates test runs between groups of agents (not an endpoint for any testing)
- Manages agent registration and remediation
- Interacts with databases
- Manages group topology
- Provides HTTP API to the ToDD client and 3rd party software

3.2 ToDD Agent

The Agent is actually where the test is run. The specific tasks that a particular agent must perform are calculated by the server in order to facilitate the "big picture" of the overall system test, and pushed down to the agent via the Comms abstraction.

Here are some specific notes on the agents.

- Provides facts about operating environment back to server
- Receives and executes testrun instructions from server
- Variety of form factors (baremetal, VM, container, RasPi, network switch)

3.3 ToDD Client

The ToDD client is provided via the "todd" shell command. Several subcommands are available here, such as "todd agents" to print the list of currently registered agents.

- Manages installed ToDD objects (group and testrun definitions, etc)
- Queries state of ToDD infrastructure ("todd agents", "todd groups", etc.)
- Executes testruns ("todd run . . . ")

CHAPTER 4

Install and Run

4.1 Compile from Source

First, make sure the following software is installed and correctly configured for your platform:

- Go (at least 1.6)
- Ensure `$GOPATH\bin` is also in your `$PATH`
- Git

Note: If you are installing ToDD on a Raspberry Pi, there are specialized install instructions here.

The best way to install ToDD onto a system is with the provided Makefile. In this section, we'll retrieve the ToDD source, compile into the three ToDD binaries, and install these binaries onto the system.

First, `go get` the ToDD source. As mentioned at the beginning of this document, this assumes a system where Go has been properly set up:

```
go get -d github.com/toddproject/todd
```

Note: At this point, you may get an error along the lines of "no buildable GO source files in…". Ignore this error; you should still be able to install ToDD.

Navigate to the directory where Go would have downloaded ToDD. As an example:

```
cd $GOPATH/src/github.com/toddproject/todd
```

Finally, compile and install the binaries:

```
make
sudo -E make install
```

4.2 Using ToDD in a Container or Virtual Machine

4.2.1 Docker

If you instead wish to run ToDD inside a Docker container, you can pull the current image from Dockerhub:

```
mierdin@todd-1:~$ docker pull toddproject/todd
mierdin@todd-1:~$ docker run --rm toddproject/todd todd -h
NAME:
   todd - A highly extensible framework for distributed testing on demand

USAGE:
   todd [global options] command [command options] [arguments...]

VERSION:
   v0.1.0
......
```

All three ToDD binaries (as well as native testlets) are distributed inside this container and can be run as commands on top of the "docker run" command:

- `todd` - the CLI client for ToDD
- `todd-server` - the ToDD server binary
- `todd-agent` - the ToDD agent binary

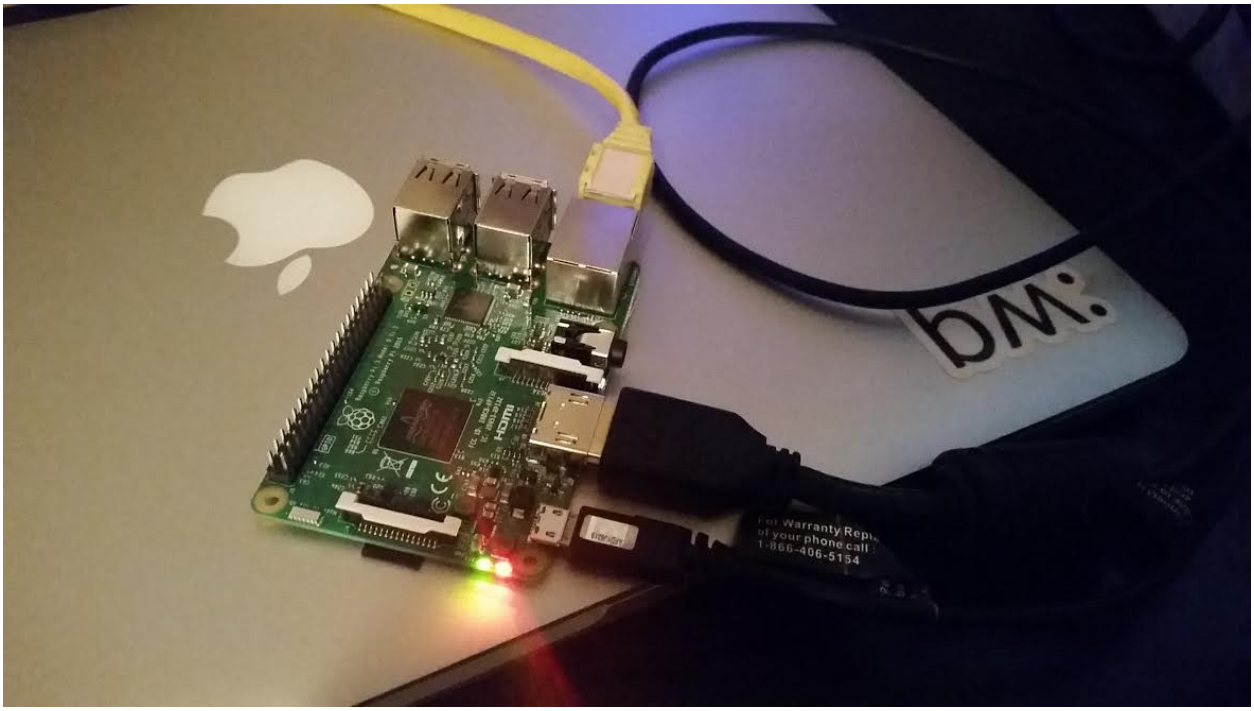
A Dockerfile is provided in the repository if you wish to build the image yourself. The Docker image repository is configured to automatically build the image from this Dockerfile whenever changes are pushed to the *master* branch in Github, so you always know you're pulling down the latest and greatest.

4.2.2 Vagrant

There is also a provided vagrantfile in the repo. This is not something you should use to actually run ToDD in production, but it is handy to get a quick server stood up, alongside all of the other dependencies like a database. This Vagrantfile is configured to use the provided Ansible playbook for provisioning, so in order to get a nice ToDD-ready virtual machine, one must only run the following from within the ToDD directory:

```
vagrant up
```

4.3 Installing on Raspberry Pi



Important: These instructions were tested on several Raspberry Pi 3 (Model B). In theory, these instructions should work for other models, but may require some minor modifications.

4.3.1 Housekeeping

In case you're new to Raspberry Pis, you can use these instructions to get a basic configuration going. This section assumes you bought the SD card with NOOBS preloaded onto it (which is recommended if you're new - it greatly simplifies the OS install)

When you first boot your Raspberry Pi, you'll be presented with the NOOBS setup. Follow the install guide, selecting "raspbian" as your operating system.

After the install finishes and you reboot your RPi, open the Raspberry Pi configuration from the button in the top left part of the screen. It's a good idea to edit a few of these. Here are some good ideas:

- Boot to CLI, not desktop
- Set hostname to something useful
- Uncheck auto login
- Change the default password for the "pi" account

Finally, open the terminal and run `ip addr show eth0` to grab the MAC address of your Pi. It is recommended that you configure the RPi to use DHCP to derive a network address, and you can use this MAC address to reserve an IP address for the Pi on your DHCP server.

Now, reboot one more, and you can finish the rest of this guide purely over SSH. You should only need a network connection and power for the rest of these steps.

4.3.2 Install Go

This install guide will use Go 1.6, but other than that, channels the excellent Go 1.5 on RPi install instructions by Dave Cheney.

An abbreviated version of those instructions, modified for Go 1.6, is shown below:

```
cd $HOME
mkdir workspace
curl https://storage.googleapis.com/golang/go1.6.src.tar.gz | tar xz
curl http://dave.cheney.net/paste/go-linux-arm-bootstrap-c788a8e.tbz | tar xj
cd $HOME/go/src
GO_TEST_TIMEOUT_SCALE=10 GOROOT_BOOTSTRAP=$HOME/go-linux-arm-bootstrap ./all.bash
export GOPATH="$HOME/workspace/"
export PATH="$PATH:$HOME/go/bin/"
```

I also added these to my .bashrc:

```
alias ll='ls -lha'
PATH=$PATH:$HOME/go/bin
GOPATH=$HOME/workspace
```

4.3.3 Install ToDD

From now on, you can follow the canonical install instructions to install ToDD. As a reminder, you first retrieve the ToDD source using `go get`, then navigate to that directory:

```
go get -d github.com/toddproject/todd
cd $GOPATH/src/github.com/toddproject/todd
```

Next, use make to compile ToDD. Be patient, the first step can take a while on the Raspberry Pi. (On the RPi3 this can be up to 15 mins!)

```
make
sudo make install
```

Note: On some versions of Go, you may need to set the environment variable "GOARM" to "5".

4.3.4 Run ToDD Agent

It's likely that you'll want to run the ToDD agent as a service. Use this systemd unit file as a starting example, and don't forget to enable the service so that it starts automatically at boot!

```
pi@todd-pi-01:~ $ cat /etc/systemd/system/todd-agent.service
[Unit]
Description=ToDD Agent

[Service]
ExecStart=/usr/bin/todd-agent

[Install]
WantedBy=network.target
```

Next, enable and start the service, and check the status to ensure it's running without issues. You should see something like this:

```
pi@todd-pi-01:~ $ sudo systemctl enable todd-agent
pi@todd-pi-01:~ $ sudo systemctl start todd-agent.service
pi@todd-pi-01:~ $ sudo systemctl status todd-agent
 todd-agent.service - ToDD Agent
   Loaded: loaded (/etc/systemd/system/todd-agent.service; static)
   Active: active (running) since Thu 2016-04-21 08:13:58 UTC; 33min ago
 Main PID: 8532 (todd-agent)
   CGroup: /system.slice/todd-agent.service
           └─8532 /usr/bin/todd-agent

Apr 21 08:47:16 todd-pi-02 todd-agent[8532]: time="2016-04-21T08:47:16Z" level=info␣
↪msg="AGENTADV -- 2016-04-21 08:47:16.577100389 +0000 UTC"
Apr 21 08:47:16 todd-pi-02 todd-agent[8532]: time="2016-04-21T08:47:16Z" level=debug␣
↪msg="Retrieving value of key - unackedGroup"
Apr 21 08:47:18 todd-pi-02 todd-agent[8532]: time="2016-04-21T08:47:18Z" level=debug␣
↪msg="Retrieving value of key - unackedGroup"
Apr 21 08:47:20 todd-pi-02 todd-agent[8532]: time="2016-04-21T08:47:20Z" level=debug␣
↪msg="Retrieving value of key - unackedGroup"
Apr 21 08:47:21 todd-pi-02 todd-agent[8532]: time="2016-04-21T08:47:21Z" level=debug␣
↪msg="Agent task received: {\"type\":\"SetGroup\",\"groupName\":\"rpi\"}"
Apr 21 08:47:21 todd-pi-02 todd-agent[8532]: time="2016-04-21T08:47:21Z" level=debug␣
↪msg="Retrieving value of key - group"
Apr 21 08:47:21 todd-pi-02 todd-agent[8532]: time="2016-04-21T08:47:21Z" level=info␣
↪msg="Already in the group being dictated by the server: rpi"
Apr 21 08:47:22 todd-pi-02 todd-agent[8532]: time="2016-04-21T08:47:22Z" level=debug␣
↪msg="Retrieving value of key - unackedGroup"
Apr 21 08:47:24 todd-pi-02 todd-agent[8532]: time="2016-04-21T08:47:24Z" level=debug␣
↪msg="Retrieving value of key - unackedGroup"
Apr 21 08:47:26 todd-pi-02 todd-agent[8532]: time="2016-04-21T08:47:26Z" level=debug␣
↪msg="Retrieving value of key - unackedGroup"
```

4.4 Configuring ToDD

ToDD uses configuration files (typically found in `/etc/todd`) to control its behavior. The server and the agent use their own individual configuration files (examples of each are shown below).

Note: Configuration files for both `todd-server` and `todd-agent` binaries load once at initial start. So, in order to apply any changes to the configuration, these processes need to be restarted.

4.4.1 Server Configuration

The server configuration file (usually `/etc/todd/server.cfg`) contains configurations for all services that the server will require - things like integrated databases, communications plugins, and internal calculations.

Warning: Pay particular note to the `LocalResources` section of the configuration. The `DefaultInterface` option is required (or in lieu thereof, the `IPAddrOverride` option) so that the server will know what IP address to serve assets from to the agents. The server will not start if it cannot determine its IP address using these options.

If this is configured incorrectly, agents will be unable to retrieve the required asset files, and will fail to register with the server.

Here is a sample server configuration file, with comments inline:

```
# ToDD's API
[API]
Host = 0.0.0.0
Port = 8080

# Serves assets like collectors, testlets, etc.
[Assets]
IP = 0.0.0.0
Port = 8090

# Describes parameters for the "comms" system, which manages communications between
# the server and the agents
[Comms]
User = guest                        # Username for comms
Password = guest                    # Password for comms
Host = localhost                    # Hostname or IP address for comms
Port = 5672                         # Port for comms
Plugin = rabbitmq                   # Comms plugin to use (i.e. "rabbitmq")

# Parameters for database connectivity
[DB]
Host = 192.168.0.10                   # Hostname or IP address for database
Port = 4001                         # Port for database
Plugin = etcd                       # Database plugin to use (i.e. "etcd")

# Parameters for time-series database connectivity
[TSDB]
Host = 192.168.0.10                   # Hostname or IP address for tsdb
Port = 8086                         # Port for tsdb
Plugin = influxdb                   # TSDB plugin to use (i.e. "influxdb")

[Grouping]
Interval = 10                       # Interval (in seconds) for the grouping␣
↪calculation
                                    # to run on the server

[Testing]
Timeout = 30                        # This is the timer (in seconds) that a test will␣
↪be
                                    # allowed to live

# Describes parameters for local resources, such as network or filesystem resources
[LocalResources]
DefaultInterface = eth2             # Dictates what network interface is used for␣
↪testing
                                    # purposes (i.e. informs the todd-server which IP
                                    # address can be used

IPAddrOverride = 192.168.99.100     # Overrides DefaultInterface by providing a␣
↪specific IP
                                    # address rather
```

(continues on next page)

(continued from previous page)

```
OptDir = /opt/todd/agent            # Operational directory for the agent. Houses␣
↪things like
                                    # cache files, user-defined testlets, etc.
```

4.4.2 Agent Configuration

The agent configuration (usually `/etc/todd/agent.cfg`) is considerably simpler than the server configuration.

Warning: Similar to the server configuration, the `LocalResources` section is very important. The `DefaultInterface` option is required (or in lieu thereof, the `IPAddrOverride` option) so that the agent can report a usable address back to the server in order for it to facilitate tests. Like the ToDD server, the agent will not start if it cannot determine its IP address.

Again, comments are provided below to help illustrate the various options:

```
# Describes parameters for the "comms" system, which manages communications between
# the server and the agents
[Comms]
User = guest                        # Username for comms
Password = guest                    # Password for comms
Host = localhost                    # Hostname or IP address for comms
Port = 5672                         # Port for comms
Plugin = rabbitmq                   # Comms plugin to use (i.e. "rabbitmq")

# Describes parameters for local resources, such as network or filesystem resources
[LocalResources]
DefaultInterface = eth2             # Dictates what network interface is used for␣
↪testing
                                    # purposes (i.e. informs the todd-server which IP
                                    # address can be used

IPAddrOverride = 192.168.99.100     # Overrides DefaultInterface by providing a␣
↪specific IP
                                    # address rather

OptDir = /opt/todd/agent            # Operational directory for the agent. Houses␣
↪things like
                                    # cache files, user-defined testlets, etc.
```

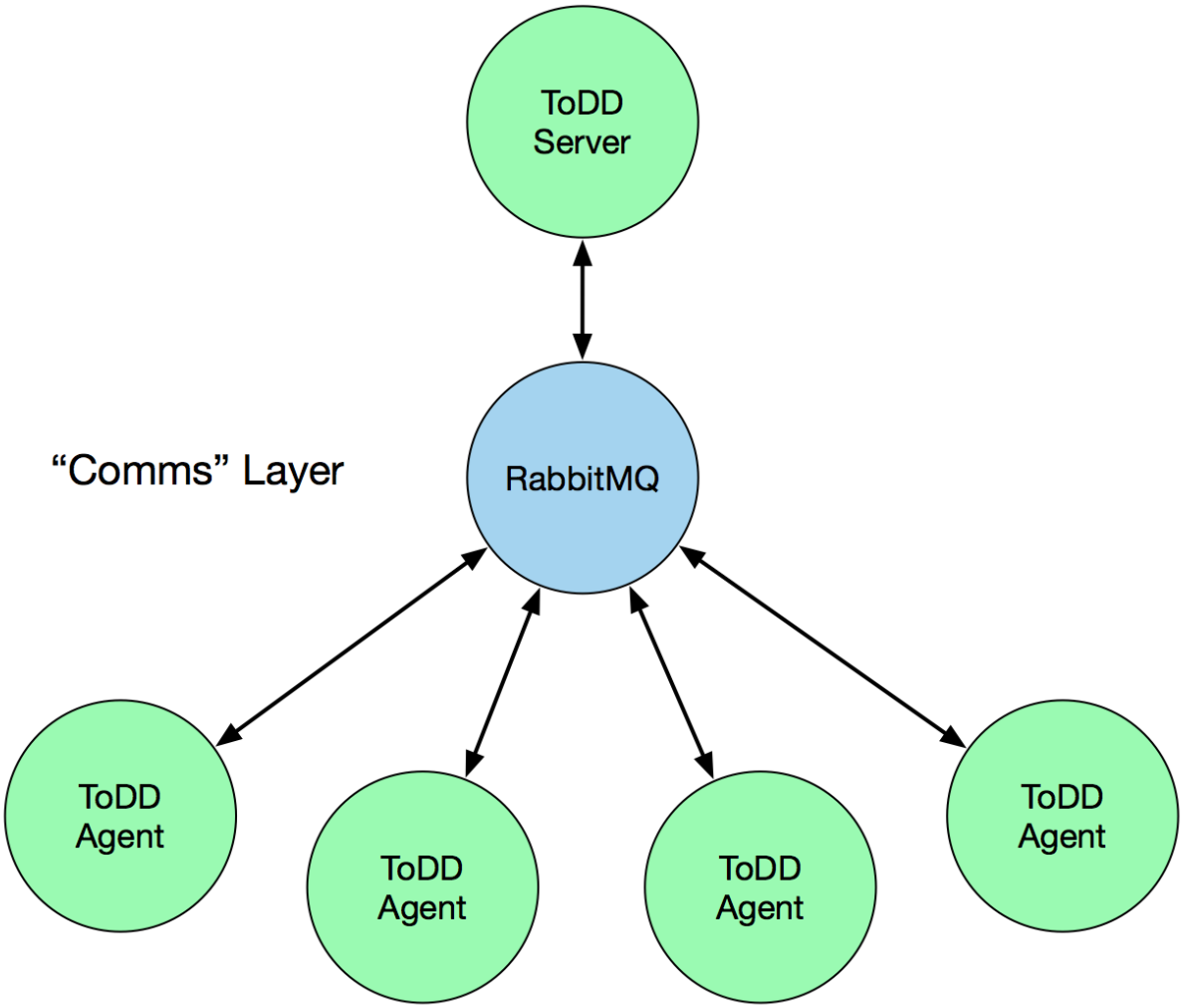
These instructions will help you get ToDD running on your system. However, keep in mind that once you've done this, there are also some external dependencies that must also be provided.

CHAPTER 5

Dependencies

5.1 Agent Communication (Comms)

In ToDD, we use a simple abstraction for communicating between the ToDD server and the agents, and this is implemented in the "comms" package. This abstraction allows ToDD to easily work with external messaging services like RabbitMQ, or Apache Kafka, etc - in order to facilitate distributed communications. As a result, ToDD agents do not actually speak directly with the ToDD server - they communicate indirectly through this third-party entitity.



As shown in #18, this paradigm wasn't very well documented until now. ToDD still requires that some external entity like RabbitMQ is set up.

If you have no experience with doing this, please refer to start-containers.sh - this starts some docker containers running these services with reasonable defaults - suitable for getting started with ToDD.

5.1.1 Comms

The "comms" abstraction within ToDD integrates with RabbitMQ (and others TBD) to provide this communication.

Note: The rest of this doc is primarily focused on developers looking to augment ToDD, either by adding a new comms plugin, or perhaps modifying an existing one.

Within this package, there is a file "comms.go" which contains little more than an interface and a base struct that all comms plugins must follow. In order to be considered a comms plugin, an implementation must satisfy the interface described there. This interface describes functions like AdvertiseAgent(), ListenForTasks(), and more. Through this, all plugins must implement the same behavior, and in theory, any comms plugin can be used to facilitate server-to-agent communications.

The rest of this documentation will describe the behind-the-scenes behavior of the comms plugins currently implemented within ToDD.

5.1.2 RabbitMQ

The RabbitMQ plugin is the first comms plugin to be implemented within ToDD. This plugin uses a fairly simple model of communicating with agents, and while scale was and is an important goal for the ToDD project, the RabbitMQ plugin was designed primarily for ease of use.

5.2 State Database

TBD

5.3 Time-Series Database

TBD

5.4 Internal Dependencies

Internal dependencies, such as Go libraries that ToDD uses to communicate with a message queue, or a database for example, are vendored in the `vendor` directory of the repository. There is no additional step to download such dependencies, in order to install ToDD from source and run it.

5.5 External Dependencies

There are a number of external services that ToDD needs in order to run.

- Agent Communications
- State Database
- Time-Series Database

Please refer to the specific pages linked above for each to see what specific integrations have been built into ToDD in each area.

CHAPTER 6

User Guide

6.1 Working with the ToDD Client

ToDD comes with an easy-to-use CLI client. Use the `--help` flag to print some useful help output:

```
mierdin@todd-1:~$ todd --help
NAME:
   todd - A highly extensible framework for distributed testing on demand

USAGE:
   todd [global options] command [command options] [arguments...]

VERSION:
   v0.1.0

COMMANDS:
   agents   Show ToDD agent information
   create   Create ToDD object (group, testrun, etc.)
   delete   Delete ToDD object
   groups   Show current agent-to-group mappings
   objects  Show information about installed group objects
   run      Execute an already uploaded testrun object
   help, h  Shows a list of commands or help for one command

GLOBAL OPTIONS:
   -H, --host "localhost"   ToDD server hostname
   -P, --port "8080"        ToDD server API port
   --help, -h           show help
   --version, -v        print the version
```

6.1.1 Agents

Use the `todd agents` command to display information about the agents currently known to the ToDD server.

```
mierdin@todd-1:~$ todd agents --help
NAME:
   temp-client agents - Show ToDD agent information

USAGE:
   temp-client agents [arguments...]
```

You can run this command on it's own to display a summary of all agents, their facts, collectors, etc:

```
mierdin@todd-1:~$ todd agents
UUID          EXPIRES ADDR         FACT SUMMARY         COLLECTOR SUMMARY
4c1ef1fd94ce  23s     172.18.0.7   Addresses, Hostname get_addresses, get_hostname
cba4e720efae  24s     172.18.0.8   Addresses, Hostname get_addresses, get_hostname
555dacccb4ae  24s     172.18.0.9   Addresses, Hostname get_addresses, get_hostname
79ffae90354e  24s     172.18.0.10 Hostname, Addresses get_addresses, get_hostname
42b1341c22fe  24s     172.18.0.11 Addresses, Hostname get_addresses, get_hostname
fdb4c3ddc8eb  25s     172.18.0.12 Addresses, Hostname get_hostname, get_addresses
```

Or, you could append an agent UUID to this command to see detailed information about that agent, such as the facts that it is reporting:

```
mierdin@todd-1:~$ todd agents 4c1ef1fd94ce
Agent UUID:  4c1ef1fd94ce91c9c589880c47fb5374bba91ecdeb852a9ac3bb4278507c0ba4
Expires:  25s
Collector Summary: get_addresses, get_hostname
Facts:
{
    "Addresses": [
        "127.0.0.1",
        "::1",
        "172.18.0.7",
        "fe80::42:acff:fe12:7"
    ],
    "Hostname": [
        "todd-agent-0"
    ]
}
```

6.1.2 Create

Important: See the documentation on ToDD objects for an explanation on what they are and how they work.

Use the `todd create` command to upload an object to the ToDD server.

```
mierdin@todd-1:~$ todd agents --help
NAME:
   temp-client agents - Show ToDD agent information

USAGE:
   temp-client agents [arguments...]
```

6.1.3 Delete

Important: See the documentation on ToDD objects for an explanation on what they are and how they work.

Run "todd delete"

6.1.4 Groups

Run "todd create"

6.1.5 Objects

Important: See the documentation on ToDD objects for an explanation on what they are and how they work.

Run "todd objects <type>"

6.1.6 Run

Run "todd create"

Show optional arguments

6.2 ToDD Objects

There are a number of concepts that power ToDD. If you haven't read that document, please take the time to review that now. In ToDD, the usage of these concepts is defined YAML files. These are imported into ToDD and are generally referred to as "objects". However, each concept is associated with its own object "type".

This document will outline the required syntax for these YAML definitions, as well as the commands needed to use these with ToDD.

6.2.1 All Objects

There are a few components that must be included in every ToDD object definition. These are fields you'll notice regardless of the type of object being defined. They are:

- `type` - the type of object being defined (i.e. "group", or "testrun")
- `label` - human-readable name for this object so that you can refer to it later
- `spec` - the details of the object definition

The "meat" of any ToDD object definition will be found underneath `spec`, as that is where the type-specific details will be found. For instance, a group definition's matching statements will all appear here.

Note: This page is intended to illustrate the syntax for ToDD objects, not how to work with the ToDD CLI to create, delete, or retrieve objects. For more information on that, please refer to the CLI reference.

6.2.2 Group

To review, a "group" is a collection of agents with similar attributes. The YAML files that define groups can use match statements to identify which agents fall into that group, based on the attributes collected about that node by the ToDD agent.

The core of a group definition is the list of "match" statements. These statements are evaluated one at a time, from first, to last when evaluating an agent. If a statement matches an agent attribute, the rest of the statements are skipped, and the agent becomes part of that group. If no items match, the agent is not part of that group, and other group objects are considered. If no match statements in any group definition matches an agent, the agent remains ungrouped, and therefore cannot be used for testing.

A match statement can allow you to group agents in one of two ways:

- Hostname
- IP Subnet

First, let's look at a group definition that groups three agents via hostname:

```
---
type: group
label: group-datacenter
spec:
    group: datacenter
    matches:
    - hostname: "todd-agent-1"
    - hostname: "todd-agent-2"
    - hostname: "todd-agent-3"
```

In the above example, only three agents will be placed into this group, as their hostname is being matched directly using one of the three match statements.

However, these `hostname` statements can also use regular expressions, to help simplify the group definition by allowing a statement to potentially match multiple agents. The below example is equivalent to the previous definition, but uses regular expression to simplify things:

```
---
type: group
label: group-datacenter
spec:
    group: datacenter
    matches:
    - hostname: "todd-agent-[1-3]"
```

Note: ToDD's grouping logic uses the regexp package in Go's standard library to compile each parameter in a `hostname` statement, so you should look there for implementation details.

In addition to matching on hostname, group definitions can also match on IP subnet. See the below example for a group definition that includes all agents in the `192.168.0.0/24` subnet:

```
---
type: group
label: group-datacenter
spec:
    group: datacenter
```

(continues on next page)

(continued from previous page)

```
    matches:
    - within_subnet: "192.168.0.0/24"
```

Note: ToDD's grouping logic uses the net.ParseCIDR function to parse the provided subnet, and the net.Contains function to determine if an agent's IP address is within that network. Please refer to the documentation for those functions for details on how they work.

6.2.3 Testrun

A `testrun` object defines the parameters for a test. Just like any other ToDD object, they must have a `type` field (set to "testrun" in this case), and a `label` field. The `spec` section contains a few fields that determine how the test will operate.

The first field is called `targettype`. This determines if the test is being run against a list of "dumb" endpoints, or a group of ToDD agents. This will depend on the testlet being used. This needs to be set to "uncontrolled" if you're just testing against one or more endpoints that aren't running ToDD agents. If your target is another ToDD group, this needs to be set to "group".

The second field is called `source`, and since all tests **originate** from a ToDD group (even though the destination may or may not be a group), we need to tell ToDD how to instruct the agents in this group to work. So, within the `source` configuration, we specify `name`, which indicates the agent group the test should originate from, `app`, which is the name of the testlet to use in this group, and `args`, which is a string of additional command line parameters that may be required by the testlet.

Finally, we also need to provide `target`. Since `targettype` was "uncontrolled", this is a list of IP addresses, or hostnames/FQDNs to test against.

Here's a working example of this kind of testrun with inline comments:

```
---
type: testrun
label: test-ping-dns-hq
spec:
    targettype: uncontrolled    # Is the test being run against
                                # ToDD agents or "dumb" nodes?
    source:
        name: headquarters      # Which agent group is the "source" for this test?
        app: ping               # What testlet should this group use for this test?
        args: "-c 10"           # Additional arguments to pass to testlet

    target:                     # Since targettype is "uncontrolled", this
                                # is a list of IP addresses or FQDNs
    - 4.2.2.2
    - 8.8.8.8
```

Testruns can also be run against other ToDD groups. For instance, the `iperf` testlet requires both a client and a server component. So, in this case, we set `targettype` to "group", and instead of a list of IP addresses under `target`, we instead provide the same three parameters that we did for `source` (though the actual values will probably be different between `source` and `target`).

Again, here's a working example of this.

```
---
type: testrun
label: test-dc-hq-bandwidth
spec:
    targettype: group           # Is the test being run against
                                # ToDD agents or "dumb" nodes?
    source:
        name: datacenter        # Which agent group is the "source" for this test?
        app: iperf              # What testlet should be used for this test?
        args: "-c {{ target }}" # Additional arguments to pass to testlet
    target:
        name: headquarters      # Which agent group is the "target" for this test?
        app: iperf              # What testlet should this group use for this test?
        args: "-s"              # Additional arguments to pass to testlet
```

CHAPTER 7

Testlets

7.1 Native Testlets

There are a number of testlets that are developed as part of the overall ToDD Project. This was done to provide immediate functionality for the vast majority of users that are looking for simple tests, that are useful to anyone. Things like ping, basic HTTP testing, and bandwidth testing are things that any user can take advantage of. These testlets are called "native testlets".

Native Testlets are maintained in their own separate repositories but are distributed alongside ToDD itself.

7.1.1 ping

[Visit Github Repository]

The `ping` testlet provides basic ICMP echo tests. It reports on things like latency, and packet loss.

Input

- The `-c` argument is used to indicate how many ICMP echos should be sent. For instance, `toddping <target> -c 5` will send 5 echo requests. If this argument is omitted, the testlet will default to 3 (note that all metrics are averaged over the number of requests)
- The `-t` argument is used to indicate the timeout value for a single request (in seconds). For instance, `toddping <target> -t 1` will set the timeout to 1 second. If the argument is omitted, this value will default to 3 seconds.

Output

Here is a sample output from the ping testlet:

{ "avg_latency_ms": "27.007", "packet_loss_percentage": "0"

}

- avg_latency_ms: This is a float value indicating the average latency for all responses
- packet_loss_percentage: a float value between 0 and 1 indicating the packet loss for the entire test

Special Considerations

On Linux, the ability to leverage ICMP sockets in software usually requires special permissions. The easiest answer is to run such software as root, or with equivalent permissions.

However, this carries it's own complexity - so there are two alternatives to running the `ping` testlet:

- A 2011 commit to the Linux kernel introduced a new ICMP socket type. Using this socket does not require root, but only if the system is configured to do so by configuring `net.ipv4.ping_group_range` with `sysctl`. This turned out to be a problem when running ToDD in a Docker container, as in this case, the kernel is shared, and would require host configuration changes. Not exactly an ideal solution, especially in cloud deployments.
- The `setcap` command can be used to provide "special" permissions to an executable binary called "capabilities". One capability, "cap_net_raw", allows applications to use raw sockets without root. This is not a system-wide setting, but rather is granted to a specific application. The `scripts/set-testlet-capabilities.sh` script, which is invoked when running `make install` to build ToDD from source, grants the `ping` testlet this capability, as well as any other native testlet that might need it.

The second option was chosen, as it was a simpler and more secure option, and one that worked on a variety of platforms.

The `ping` testlet will first try to use ICMP raw sockets, and then will fall back to using UDP. This allows it to work on both OSX (Darwin) and Linux.

7.1.2 http

[Visit Github Repository]

This page is TBD, as this native testlet has not yet begun active development. Once development efforts begin, this page will be updated as needed.

Input

TBD

Output

TBD

7.1.3 bandwidth

[Visit Github Repository]

This page is TBD, as this native testlet has not yet begun active development. Once development efforts begin, this page will be updated as needed.

Input

TBD

Output

TBD

7.1.4 portknock

[Visit Github Repository]

This page is TBD, as this native testlet has not yet begun active development. Once development efforts begin, this page will be updated as needed.

Input

TBD

Output

TBD

Native tests are written in Go for a number of reasons:

- Each testlet can leverage some common code in the ToDD repository, to keep things simple. However, each testlet is provided as a separate binary (runs as a separate process to the todd-server or todd-agent)
- Testlets can be executed consistently across different platforms. The old model of using bash scripts meant the tests had to be run on a certain platform for which that testlet knew how to parse the output

The native testlets must be installed in a location that has been added to the PATH environment variable. If you are building from source, the included Makefile will kick off some scripts that perform "go get" commands for the native testlet repositories, and if your GOPATH is set up correctly, the binaries are placed in `$GOPATH/bin`. Of course, `$GOPATH/bin` must also be in the `$PATH` environment variable, which is also a best practice for any Go project. In the future, additional installation methods should do this for you by placing all binaries in sensible locations like `/usr/local/bin`.

Note: Note that `/opt/todd/server/assets/testlets` is still used by ToDD, but only for user-defined testlets. Please see User-Defined Testlets for more information

---

If these testlets do not meet your needs, please check out the documentation for User-Defined Testlets. One of the most important design requirements for ToDD was to allow for easy introduction of user-defined testing.

7.2 User-Defined Testlets

One of the most important original design principles for ToDD was the ability for users to easily define their own testing. Indeed, this has become one of ToDD's biggest advantages over other testing solutions, both open-source and commercial.

The idea is to allow the user to use any testing application (provided it is available on the system on which the ToDD agent is running. If the user writes a script to wrap around an existing application, the testlet should handle or pass along the input/arguments as well as parse any output from the underlying application. Naturally, a testlet can perform tests itself, so the user can also write a totally self-contained testing program, provided it follows the testlet standard, which is documented below.

7.2.1 Referring to a Testlet

When you want to run a certain testlet, you refer to it by name. There are a number of testlets built-in to ToDD and are therefore reserved:

- http
- bandwidth
- ping
- portknock

Provided it has a unique name, and that it is executable (pre-compiled binary, Python script, bash script, etc.) then it can function as a testlet. Early testlets were actually just bash scripts that wrapped around existing applications like iperf or ping, and simply parsed their output.

Note: All native testlets maintain their own documentation. Please view the links at the top of Native Testlets for more information about these testlets, such as what arguments they require, and a sample of their output.

7.2.2 Check Mode

Each testlet must support a "check mode". This is sort of a "pre-test" check to ensure the testlet can be run. When running in "check mode", a testlet should test it's own ability to run a "real" test, such as sending traffic to localhost to ensure it can use the network stack.

For instance, when the ToDD agent runs the "todd-ping" testlet in check mode, it would invoke it like this:

```
todd-ping check
```

When invoked like this, any testlet should go through whatever internal checks it needs to in order to verify it can operate successfully. This means testing access to sockets, filesystem resources, etc.

When finished, and everything checks out successfully, the testlet should print the following string to stdout: "Check mode PASSED". The agent will be watching stdout for this string and will use this to know if the testlet is ready to be used.

7.2.3 Input

Obviously, testing application vary greatly in terms of their input. Some testing applications use certain command-line arguments or flags, and others aren't even configured via the command-line.

The idea of a ToDD testlet is to standardize this input so that any testing application can be run identically. This was a very useful concept early in ToDD's life, as the very first testlets were simple bash scripts that wrapped existing applications like `ping` and `iperf`, and passed around the required arguments to make them conform to the standard we'll discuss now.

All testlets must follow the following standard input:

```
./testletname < target > < args >
```

- "target" - this is always the first parameter. `todd-agent` will spin up N instances of a testlet, where N equals the number of targets that a given agent has been instructed to test against. So, the testlet must accept the target's IP address or hostname as it's first argument.

- "args" - any arguments required by the testlet. These could be arguments required by the testlet itself, or they could be arguments required by an underlying application that the testlet is wrapping. This depends on the testlet implementation.

7.2.4 Output

The output for every testlet is a single-level JSON object, which contains key-value pairs representing the metrics gathered for that testlet.

Since the ToDD agent is responsible for executing a testlet, it will watch `stdout`, which is where `todd-agent` will output this JSON object. This makes testlets a very flexible method of performing tests; since it only needs to output these metrics as JSON to stdout, the testlet can be written in any language, as long as they support the input and output standardized in this document.

A sample JSON object that the "ping" testlet will provide is shown below:

```
{
    "avg_latency_ms": 27.007,
    "packet_loss_percentage": 0
}
```

Note: The ToDD agent does not have an opinion on the values contained in the keys or values for this JSON object, or how many k/v pairs there are - only that it is valid JSON, and is a single level object (no nested objects, lists, etc). It also doesn't care about the datatype for these metrics. In this case, the first metric is a float, and the second is an integer. Both are passed as-is to the TSDB, or presented to the user.

The JSON document shown above contains the metrics for a single testlet run, which means that this is relevant to only a single target, run by a single ToDD agent. The ToDD agent will receive this output once for each target in the testrun, and submit this entire dataset up to the ToDD server via the `comms` system when finished.

The ToDD Server will also aggregate each agent's report to a single metric document for the entire testrun, so that it's easy to see the metrics for each source-to-target relationship for a testrun.

7.2.5 Installing a User-Defined Testlet

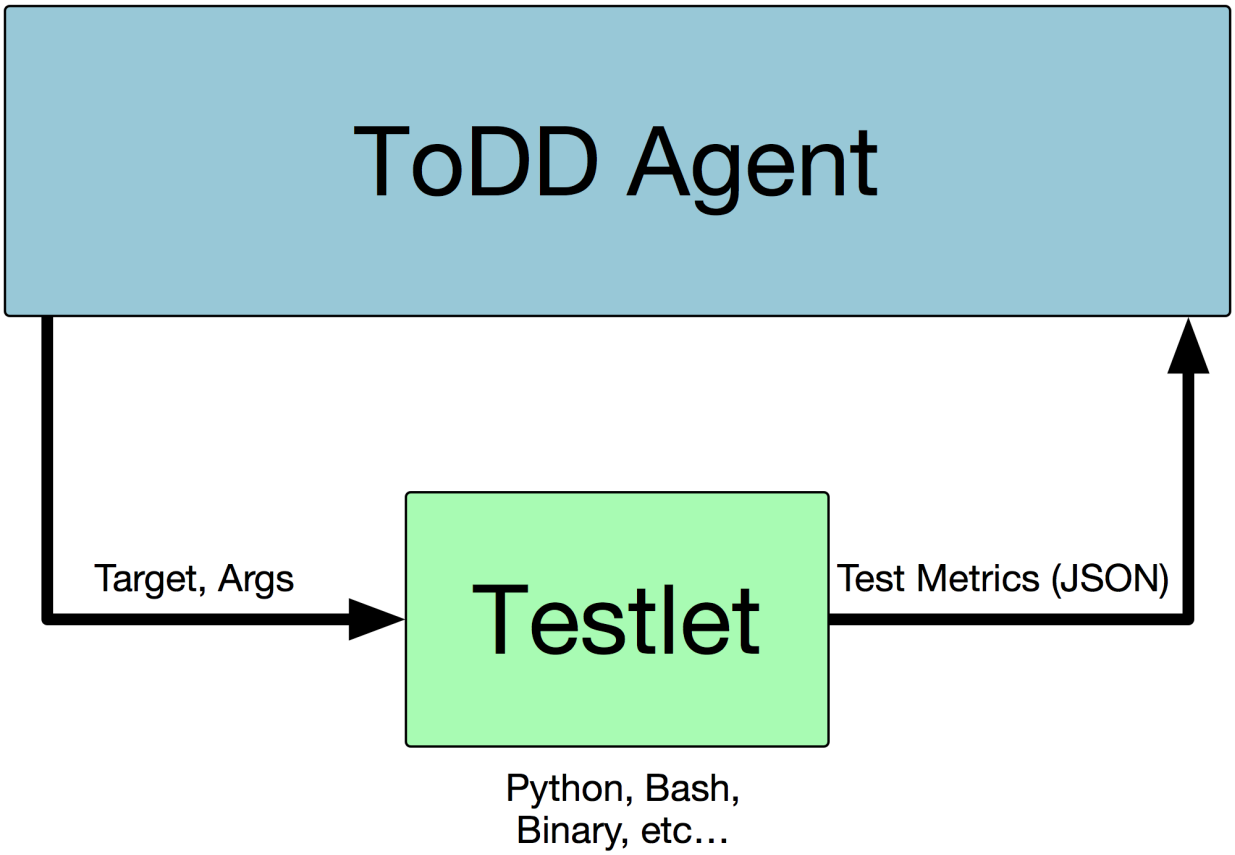
Installing a user-defined testlet is easy, once you've written it to conform to the standards above. All you need to do is place it in the directory `/opt/todd/server/assets/testlets` on the server running `todd-server`, and ToDD will ensure that the testlet is deployed to the agents.

Note: When ToDD was first released, testlets were all written in bash and packaged with ToDD by embedding them in some clever Go code, and then "unpacked" into this directory. So, you might see some testlets in this directory that you didn't write, like `iperf`, or `http`. This is a temporary measure, and will not exist in a future version of ToDD, once those testlets are implemented in Go.

If you want to make changes to these testlets, just overwrite the file in this directory - ToDD will detect those changes and continually make sure the agents are updated.

Testing applications are called "testlets" in ToDD. This is a handy way of referring to "whatever is actually doing the work of testing". This concept keeps things very neatly separated - the testlets focus on performing tests, and ToDD focuses on ensuring that work is distributed as the user directs.

Testlets are executed directly by the ToDD agent. The agent will pass a standardized set of arguments to the testlet, such as the target of the test, as well as any other useful parameters. Once finished, the testlet will then send metrics back to the agent by printing a JSON object to stdout.



This generic interface makes testing in ToDD very flexible.

There are a number of testlets (known as "native testlets") that have been developed as part of the ToDD project, because they represent very common use cases for many users:

- http
- bandwidth
- ping
- portknock

They run as separate binaries, and are executed in the same way that custom testlets might be executed, if you were to provide one. If you install ToDD using the provided instructions, these are also installed on the system.

Note: If, however, you wish to build your own custom testlets, refer to Custom Testlets; you'll find it's quite easy to build your own testlets and run them with ToDD. This extensibility was a core design principle of ToDD since the beginning of the project.

If you're not a developer, and/or you just want to USE these native testlets, you can install these binaries anywhere in your PATH. The included Makefile will do this for you (provided a proper Go setup), and future installation methods will also automate this process.

CHAPTER 8

Additional Resources

8.1 Collaboration

The most active platform at the moment is the #todd channel in the "Network to Code" slack team. This team is free and open to join - just head over to http://slack.networktocode.com/ to sign up, and look for the #todd channel once in.

ToDD also has a mailing list, for all matters related to development, or questions about getting ToDD running on your own systems. Please feel free to join and send a message if you have any questions! We will be as helpful as possible.

8.2 Blogs

I wrote a blog post on ToDD, as part of the release. Plenty of nice visuals!

8.3 Videos

ToDD was first released at the Devops Networking Forum in Santa Clara. That talk is viewable here:

I also recorded a video that covers some of the high-level concepts and design of ToDD:

Also, please see the video below for a full demonstration of ToDD in action:

CHAPTER 9

Roadmap

Goals for Alpha Release

- Please see https://github.com/toddproject/todd/milestone/1

Goals for Beta Release

- TBD

Goals for Full Release

- Web Front-End?