ping++ API

Cloudflare 2020 Internship Coding Challenge

Language Selection: C++

Zachery Utt
utt.zachery@ufl.edu
321-278-6920
University of Florida

Table of Contents

Flag Summary:

ping++ [hostname / IP] [-t timeToLive] [-o timeout] [-c numPackets] [-w waitInterval] [-f onFail] [-e errorThreshold] [-m minTTL] [-s [sampleSize]]

Flag	Description	Default Value
[hostname / IP]	Defines the host to ping. If a URL is provided, a DNS lookup is performed to generate an IPv4/IPv6 address. If an IPv4/IPv6 address is provided, it will be used directly. Throws a HostDnsException if the DNS lookup fails. Throws a SyntaxException if the input fails to match to an IPv4/IPv6 or URL regular expression	www.google.com
[-t timeToLive]	Defines a custom time to live value that will be attatched to each outgoing packet. If the timeToLive value is insufficient, a TTLExceededException exception is thrown. The value for timeToLive must be an integer, and it must be greater than zero.	255
[-o timeout]	Specifies a custom timeout to wait (in seconds) for a server response. If the socket exceeds this value, a SocketException is thrown.	5
[-c numPackets]	Specifies the number of packets that should be sent each PING request. Each packet will consist of 64 bytes, and will be sequenced with a unique identifier that must be returned, or else a SocketException occurs.	1
[-w waitInterval]	Specifies the time to wait, in miliseconds, between subsequent PING requests. If **numPackets** is greater than 1, all packets will be sent simultaneously and the **waitInterval** refers to the time between subsequent **numPackets** transmissions	5000
[-f onFail]	Specifies a command to run if an AbstractException is thrown, and if the **errorThreshold** is met. The command will be executed with the first argument representing the error code, and the second argument representing a string breakdown of the error. This is very useful for notifications of systems failure	
[-e errorThreshold]	As PING requests are completed, a rolling average of the most recent 100 attempts is calculated to measure the current state of the host system. If the rolling average falls below the **errorThreshold**, the **onFail** command is executed.	0
[-m minTTL]	Experimentally finds the minimum TTL to get a response. If this flag is used, the application will exit after the **minTTL** is found.	
[-s [sampleSize]]	Runs a sample of **sampleSize** (if provided, otherwise of size 40) PING attempts and calculates advanced sample statistics, including confidence intervals and binomial probability calculations. If the flag is used, the application will exit after the calculations are returned	40

Flag Behavior

Many of the flags included are tightly coupled by nature of a ICMP request. For instance, a valid IP flag is not easily interchangeable or extendable. However, other flags, like the **SampleFlag** are easily extendable.

All flags must adhere to the AbstractFlag interface. The interface requires that each flag support self-parsing, that is, each flag will be given all of the runtime arguments and will alter its own behavior to match what the client is requesting.

There are two types of Flags in this application: finite flags and infinite flags.

A finite flag has a specific purpose and will only execute while a particular condition is true. For instance, the **SampleFlag** is designed to PING a fixed number of times, and perform direct statistics on the sample. The AbstractFlag interface allows for such classes through the use of the optional executeCommand(void* flagFacade) interface. In other words, flags can directly implement custom functionality and gain access to the information parsed (through the FlagFacade) in all other flags simply by overriding this method.

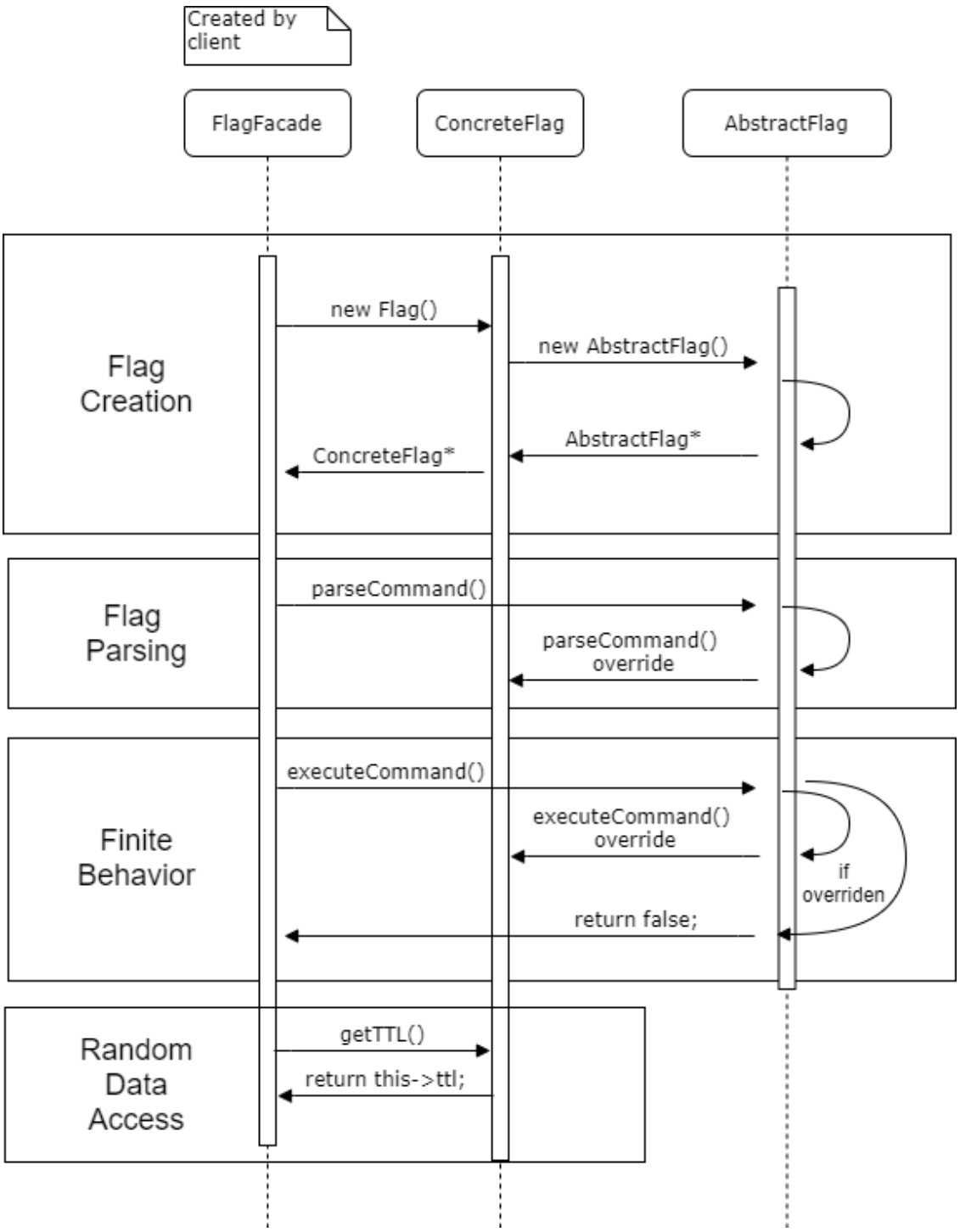
It would not make sense for the application to continue indefinitely after the information the user requested is provided, so each flag determines if execution should continue. The executeCommand(void* flagFacade) interface returns a boolean value describing whether the application should continue to run infinitely. A value of true indicates that the application should exit after all finite commands are run, and a value of false (the default value) will allow the application to proceed with infinite PING requests. It only takes one flag to return true to disable infinite PING requests.

Finite operations are performed in the order in which the flags are given. For instance, if a user initiates the **SampleFlag** before the **MinTtlFlag**, the user will receive statistics and probabilities before the user will receive the minimum experimental TTL.

Some flags do not define their own custom behavior, but rather are representations of specific data to change a different object's behavior. These flags are known as infinite flags, because they do not interfere with the applications ability to complete infinite PING transmission. For instance, the **TtlFlag** simply provides an integer representing the TTL that the user would like to use- it has no intrinsic behavior and must be used by something else to be useful.

Flag Structure

Each flag participates in a maximum of 4 stages of use, as depicted by the sequence diagram below.



All flags must adhere to steps 1 and 2 shown in the diagram: flag creation and flag parsing.

Flag Creation: Concrete Flags are created by the constructor of the **FlagFacade** object. A ConcreteFlag's constructor can accept special arguments, but should not accept references to other flags, the **FlagFacade**, or the **PingCommand** classes, as these would break encapsulation and cause nasty circular dependency.

During creation, flags should assume the user has provided no custom information and populate its fields with default values. There will be no other opportunity for the flag to do so. If a user wishes to override the default functionality with a custom flag command, this functionality will be adjusted later once the function parseCommand()is invoked.

Flag Parsing: Concrete flags must implement a custom flag parsing function. After all flags have been constructed, the **FlagFacade** will iterate over the newly created flags and supply them, through the parseCommand() interface, with the arguments that have been supplied to the application at runtime. At this time, it is safe for the flag to conduct flag specific tasks, such as the **IpFlag** invoking a DNS lookup. During parsing, flags should make use of regular expressions and throw SyntaxException where appropriate. Flags will be parsed in the order that the user supplied them, so there is no guarantee that operations conducted on other flags through the parseCommand() interface will work, as other flags may or may not have been parsed. Save all flag-interdependent tasks for the Finite Behavior phase to avoid this issue.

Finite Behavior: After construction and parsing, the **FlagFacade** will invoke the executeCommand(void* flagFacade) interface on all flags. At this time, all flags should be populated with their specific values and should have performed any runtime processing and error checking. It is now that if a custom flag wishes to perform an operation with other flags that it may do so by deserializing and casting the void* flagFacade reference. This cast should be completed in a safe, try catch block in the event an invalid pointer was supplied.

Random Data Access: Many flags, particularly infinite flags, serve as a mere representation of user data and do not add any direct functionality. These flags are useful to return custom data when prompted for dispatch by the **FlagFacade**. Any public getter functions should have a corresponding dispatch function in the **FlagFacade** class.

FlagFaçade

To manage the creation and execution of the complex interdependent flag operations, the **FlagFacade** class is responsible for creating, destroying, querying, and iterating through all Concrete Flags. The **FlagFacade** class maintains, through composition, references to all concrete flags. Further, the **FlagFacade** class adheres to the interface of getters/setters to connect, through single dispatch, clients with pertinent information.

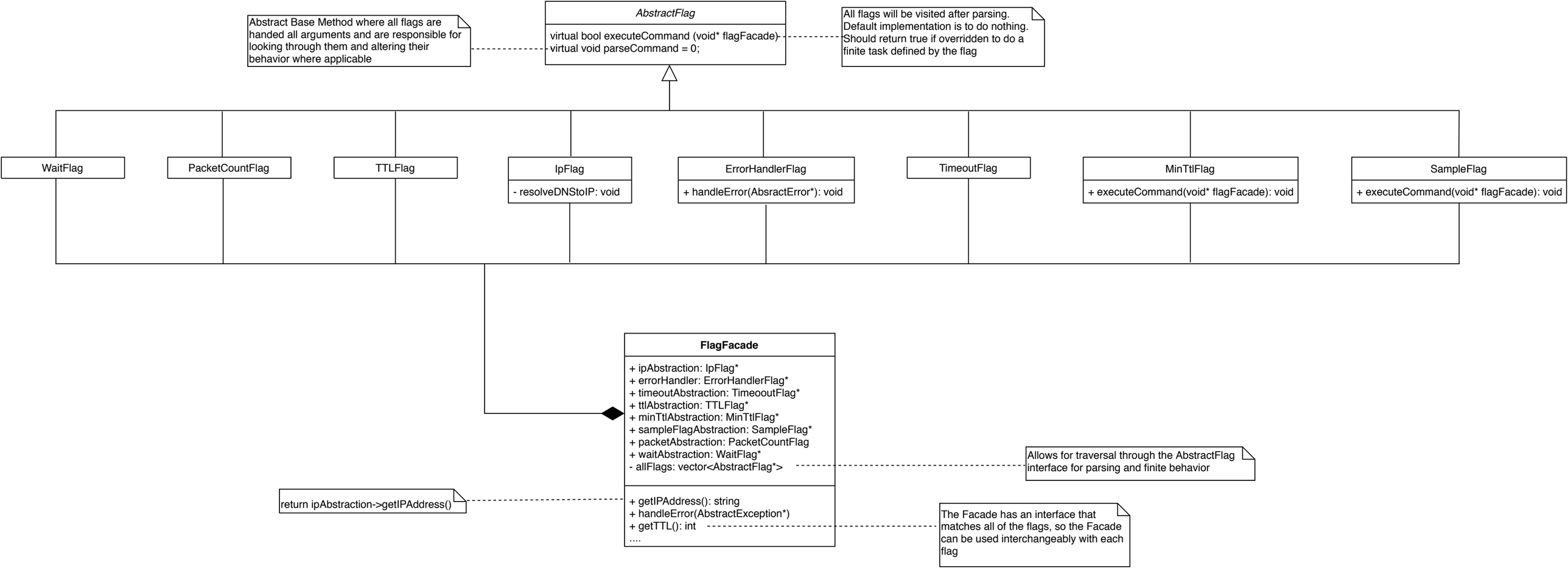
The **FlagFacade** class should not perform any additional operations while performing dispatch. It should simply deserialize the proper flag and pass the request back to the client.

Once each concrete flag is created, it is stored in the private vector<AbstractFlag*> flags member of the **FlagFacade** class. This allows the **FlagFacade** class to iterate over all concrete flags, and invoke the parseCommand() and executeCommand(void* flagFacade) functions.

To add a new concrete flag to the application, there are 4 steps that must be completed.

1. The concrete flag must adhere to the **AbstractFlag** interface
2. The concrete flag must have a reference in the **FlagFacade** class
3. The concrete flag must be added to vector<AbstractFlag*> flags
4. The **FlagFacade** class must be updated with any getters/setters within the concrete flag class through single dispatch.

Once these tasks are completed, the flag is now active.

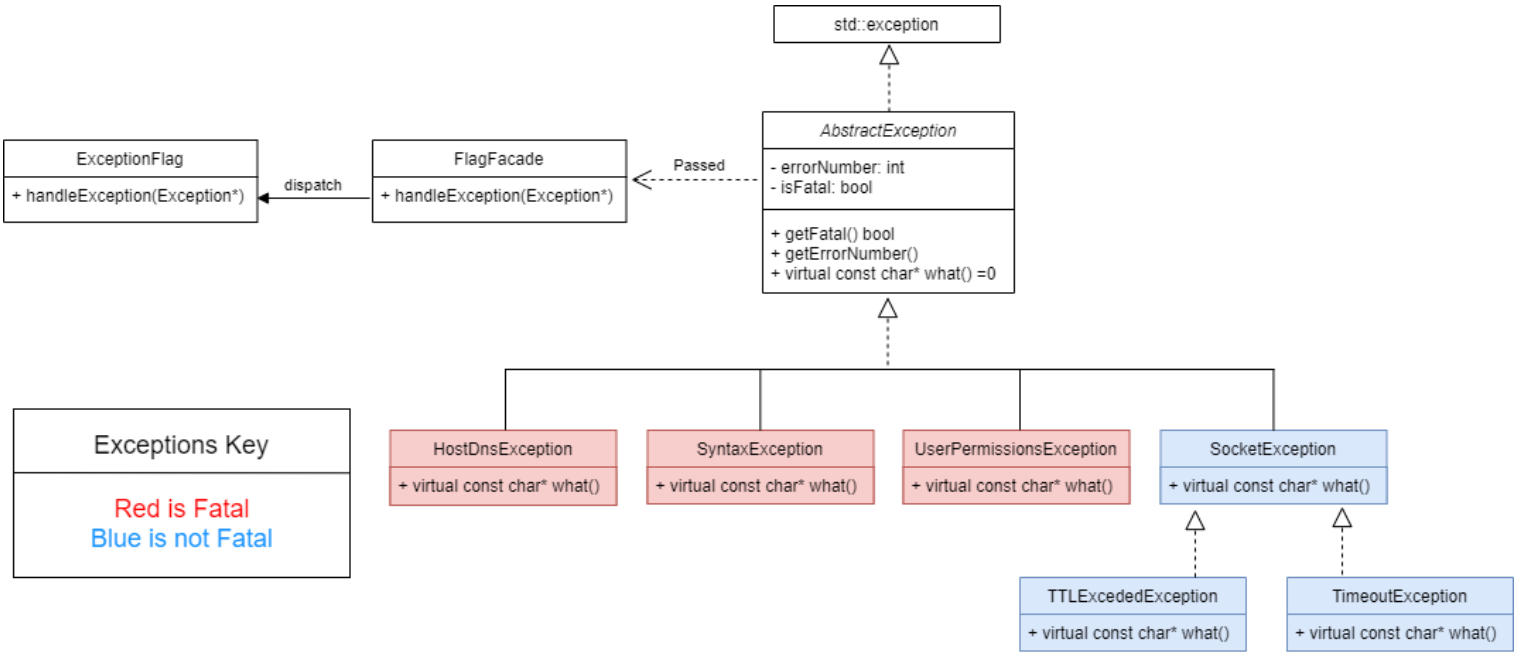

AbstractFlag
virtual bool executeCommand (void* flagFacade)
virtual void parseCommand = 0;
Abstract Base Method where all flags are handed all arguments and are responsible for looking through them and altering their behavior where applicable
All flags will be visited after parsing. Default implementation is to do nothing. Should return true if overridden to do a finite task defined by the flag
WaitFlag
PacketCountFlag
TTLFlag
IpFlag
- resolveDNStoIP: void
ErrorHandlerFlag
+ handleError(AbsractError*): void
TimeoutFlag
MinTtlFlag
+ executeCommand(void* flagFacade): void
SampleFlag
+ executeCommand(void* flagFacade): void
FlagFacade
+ ipAbstraction: IpFlag*
+ errorHandler: ErrorHandlerFlag*
+ timeoutAbstraction: TimeooutFlag*
+ ttlAbstraction: TTLFlag*
+ minTtlAbstraction: MinTtlFlag*
+ sampleFlagAbstraction: SampleFlag*
+ packetAbstraction: PacketCountFlag
+ waitAbstraction: WaitFlag*
- allFlags: vector<AbstractFlag*>
Allows for traversal through the AbstractFlag interface for parsing and finite behavior
return ipAbstraction->getIPAddress()
+ getIPAddress(): string
+ handleError(AbstractException*)
+ getTTL(): int
....
The Facade has an interface that matches all of the flags, so the Facade can be used interchangeably with each flag

Exceptions

As a networking application, exception handling is critical. Each exception thrown adheres to the custom AbstractException class. This class provides both the user and the ErrorHandlerFlag class with information on any error and its severity. Each AbstractException object contains two members: int errorNumber and bool isFatal. The isFatal member tells the ErrorHandlerFlag class that the AbstractException thrown warrants an application shutdown, and defines what system state to exit in.

If an AbstractException thrown warrants an application shutdown, and the ErrorHandlerFlag was provided by the user, then a custom command supplied to the application will be executed. Likewise, if a nonfatal exception (such as a Timeout) occurs with enough frequency, the custom command will be executed.

Success is computed through a rolling average of the general success rate. If a PING comes back successful with the correct headers or if an exception is thrown, the rolling average is adjusted. Only the past 100 PING commands are used when calculating the average success rate to best approximate the current general success rate. If this were not so, PING commands that continue indefinitely would contain a very large lag time if the host system were to fail, due to the high sample size.



Exception Descriptions

Exception	Description	Error Code
HostDnsException	Occurs when a DNS lookup could not map a host name to an active IP address. If this continues to occur, try rerunning the program with an IP address instead of a DNS address	1
SyntaxException	Occurs when a concrete flag throws an exception during parseCommand(). This could be because (1) an invalid URL was provided (2) the second value of a flag was expected, but not found (3) a flag was expecting a positive value and received a negative or zero value	2
UserPermissions Exception	Occurs when the software is not run as the root user. C sockets require root user permissions for creating sockets.	3
SocketException	General base exception that occurs when an error occurs during the transmission, reading, or writing of packets	> 3
SocketException	Memory exception that occurs when the heap can't allocate a package of 64 bytes.	4
SocketException	After indicating that the exception was valid, the operating system failed to read any bytes.	6
SocketException	The packet that the HOST returned contained too few bytes to be a valid ICMP packet	7
SocketException	The packet that the HOST returned contained the incorrect sequence identifier. If the packet count flag is provided, each packet is given a unique sequence identifier and the host must return each packet with its correct sequence identifier, or this exception will be thrown.	8
SocketException	The packet returned to the host computer wasn't initiated from this application, or the host server misprocessed this packet. Either way, this exception is thrown.	9
SocketException	The packet that the server returned isn't actually an ECHO request. The host misprocessed this packet.	10
TTLExceeded Exception: SocketException	The echo packet sent contained a critically low TTL value. This is part of the normal practice of the MinTtlFlag , but the exception will be thrown if the TTLFlag provided by the user is inadequate.	11
TimeoutException: SocketException	Occurs when a packet is expected but none arrives. This could be because of an invalid IP address or a lost packet. This can also occur if the packet count is greater than 1 and at least one packet is lost.	12

Language Selection

I chose to design this application in c++ with networking in c. The codebase intentionally does not include any libraries and runs in a very light weight package. In addition to the coding challenge, I have deployed the software to run on my Raspberry Pi. Given that the entire machine only has the capacity for 1 GB of RAM, c++ was a highly appealing choice due to its close proximity to the hardware and its strong memory control.

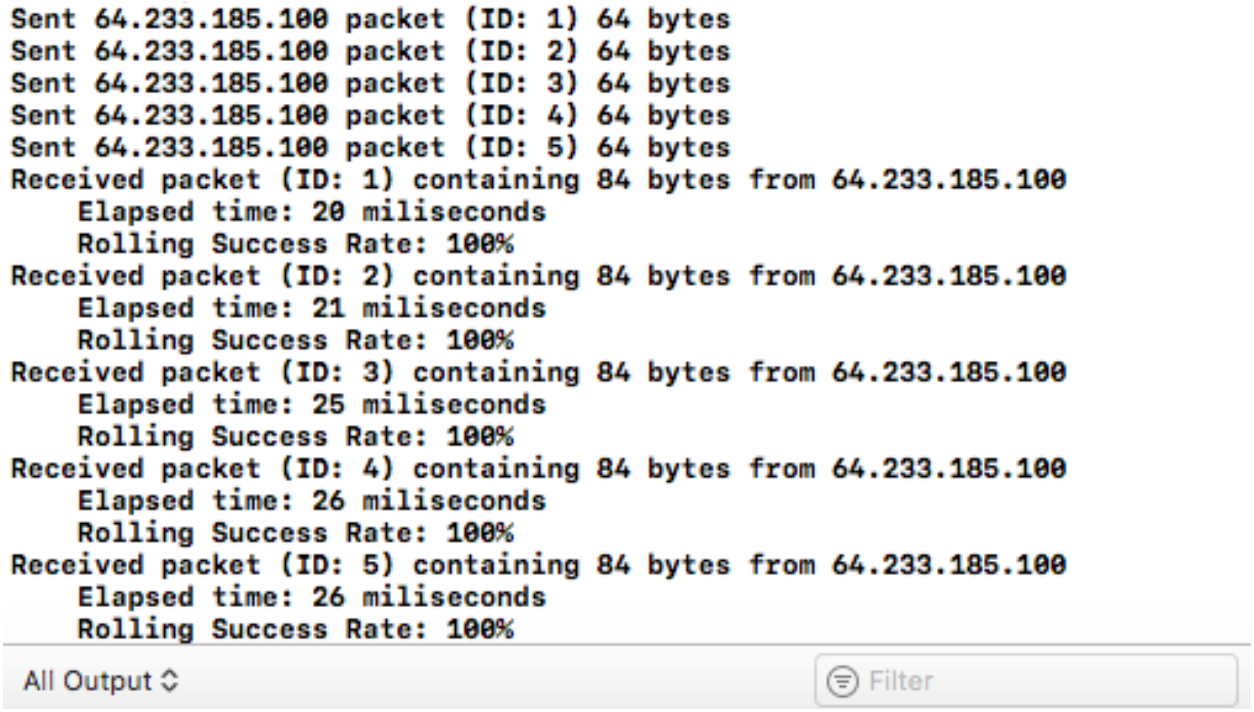
Enhancements over PING

I designed this software with one particular driving motivation: personal experience. A few years ago, I worked at a new tech startup that used an unreliable server configuration that required constant monitoring. The ping++ software offers many new features to help in this kind of scenario, including

1. Custom notifications on failing packets. By being able to provide at runtime a custom command to execute when an exception is reached or the error threshold is crossed, network administration is a whole lot easier.
2. Rolling average instead of flat average. If a PING application proceeds until interruption, a flat success average destroys the ultimate role of the application: to quickly identify and diagnose network problems. As a developer, I am not very interested in the 10,000,000 trials that worked fine a few days ago, I am interested in the trials that are currently failing. Without a rolling average, irrelevant data plagues any analysis or decision making on the part of the application.
3. Minimum TTL. The application is programmed to, when given the proper flag, experimentally deduce the minimum Time To Live value. This is valuable because developers can track how this number changes over time to gain an understanding of problems related to network infrastructure.
4. Openness for extension. With this software, it only takes 3 adjustments to the codebase to add in a new flag. Enhancements are critical for developers with custom needs, and the openness of the design allows for new custom flags and exceptions to be integrated seamlessly.
5. Error thresholding. Just because one packet failed doesn't mean that the host is experiencing an attack or a failure. By allowing the user to self-describe what the program should characterize as a critical failure,

Screenshots

Description: PING trial with a packet count of 5



Description: Advanced statistics on a sample of 40 PING trials

```
------------------------------ Summary Statistics ------------------------------

n (amount of trials): 40
p̂ (proportion of successful packet trials): 0.6
x̄ (average wait time in miliseconds): 25.2
σ² (variance of wait time in miliseconds): 72.61

------------------------------ Confidence Intervals ------------------------------

90% Confidence Interval for p: (0.472966,0.727034)
95% Confidence Interval for p: (0.448179,0.751821)
99% Confidence Interval for p: (0.400154,0.799846)
90% Confidence Interval for μ: (11.2253,39.1747)
95% Confidence Interval for μ: (8.49855,41.9015)
99% Confidence Interval for μ: (3.21543,47.1846)

------------------------------ Probabilities ------------------------------

Probability that 0% or fewer packets will be returned: 1.60694e-140
Probability that 5% or fewer packets will be returned: 3.83309e-118
Probability that 10% or fewer packets will be returned: 2.93183e-101
Probability that 15% or fewer packets will be returned: 7.93758e-87
Probability that 20% or fewer packets will be returned: 4.24596e-74
Probability that 25% or fewer packets will be returned: 1.00775e-62
Probability that 30% or fewer packets will be returned: 1.68231e-52
Probability that 35% or fewer packets will be returned: 2.63052e-43
Probability that 40% or fewer packets will be returned: 4.63109e-35
Probability that 45% or fewer packets will be returned: 1.03101e-27
Probability that 50% or fewer packets will be returned: 3.10087e-21
Probability that 55% or fewer packets will be returned: 1.29222e-15
Probability that 60% or fewer packets will be returned: 7.37846e-11
Probability that 65% or fewer packets will be returned: 5.51179e-07
Probability that 70% or fewer packets will be returned: 0.000497695
Probability that 75% or fewer packets will be returned: 0.0493533
Probability that 80% or fewer packets will be returned: 0.52819
Probability that 85% or fewer packets will be returned: 0.971723
Probability that 90% or fewer packets will be returned: 0.999955
Probability that 95% or fewer packets will be returned: 1
Probability that 100% or fewer packets will be returned: 1
Program ended with exit code: 0
```