Web Application Background

https://web.stanford.edu/class/cs114/

Philip Levis
Stanford University
pal@cs.stanford.edu

Background for Oblique

Lots of material in this lecture borrowed from CS144

Computer Systems

Computer systems focus on *abstraction*

- Software (operating systems): I have some hardware resources, what software API do I provide?
  - File systems: disk blocks become files
  - Spark: network of computers runs thousands of small tasks
- Hardware (architecture): I have digital logic, what mechanisms do I provide to software?
  - An instruction set defines how arithmetic and memory work
  - A bus defines how hardware devices can access each other

A good abstraction is easy to use and efficient while being simple to implement

Computer Systems

Computer systems focus on *abstraction*

- Software (operating systems): I have some hardware resources, what software API do I provide?
  - File systems: disk blocks become files
  - Spark: network of computers runs thousands of small tasks
- Hardware (architecture): I have digital logic, what mechanisms do I provide to software?
  - An instruction set defines how arithmetic and memory work
  - A bus defines how hardware devices can access each other

A good abstraction is easy to use and efficient while being simple to implement

Oblique

Oblique is about working within an abstraction

Particularly, how to make an abstraction (web page loading) faster despite particular architectural and security measures in place

Two key things to know:

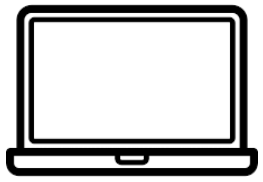
- How the web works
- What symbolic execution is

HTTP and the Web

HyperText Transfer Protocol (HTTP)

Request-response protocol that underlies the web

- Used for a lot of other things too: Google RPC
- Designed to be easy to read/understand: text



HTTP and the Web

HyperText Transfer Protocol (HTTP)

Request-response protocol that underlies the web

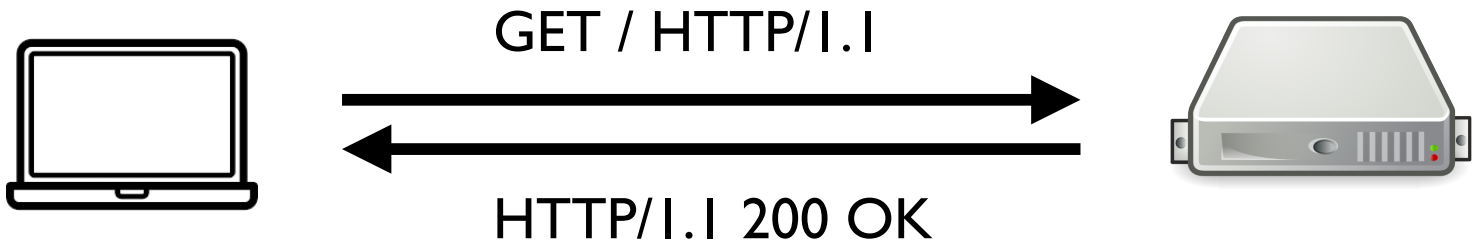
- Used for a lot of other things too: Google RPC
- Designed to be easy to read/understand: text

HTTP and the Web

HyperText Transfer Protocol (HTTP)

Request-response protocol that underlies the web

- Used for a lot of other things too: Google RPC
- Designed to be easy to read/understand: text



Networking Stack

Link Layer	Ethernet	Delivers data between computers directly connected through a medium (wire/wireless): single hop

Networking Stack

Network Layer	IP	Delivers data between computers connected across multiple hops (the Internet).
Link Layer	Ethernet	Delivers data between computers directly connected through a medium (wire/wireless): single hop

Networking Stack

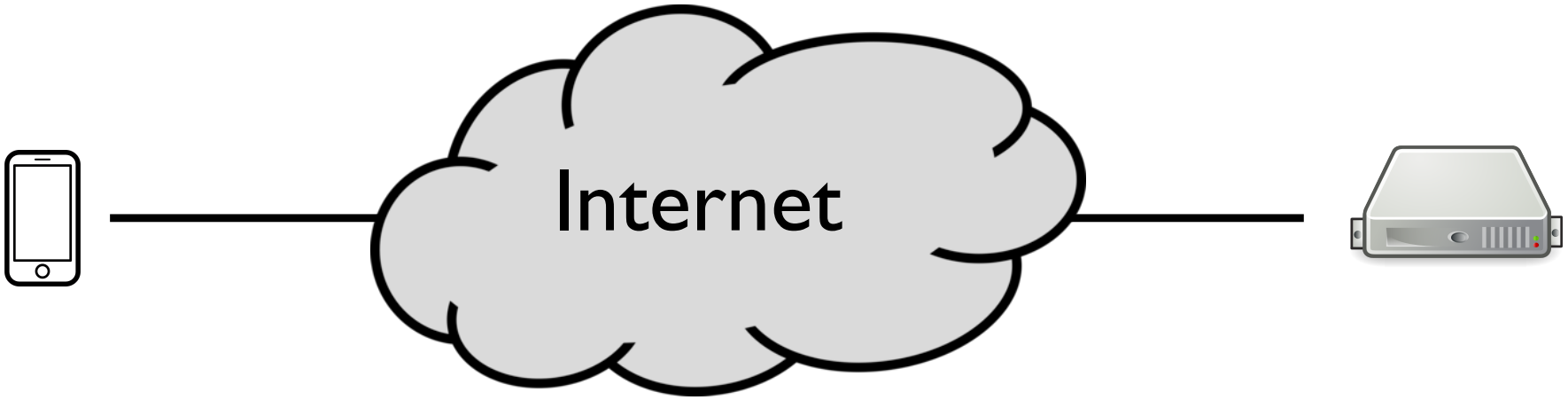
Transport Layer	TCP	Delivers data between applications (multiple applications on a computer).
Network Layer	IP	Delivers data between computers connected across multiple hops (the Internet).
Link Layer	Ethernet	Delivers data between computers directly connected through a medium (wire/wireless): single hop

Networking Stack

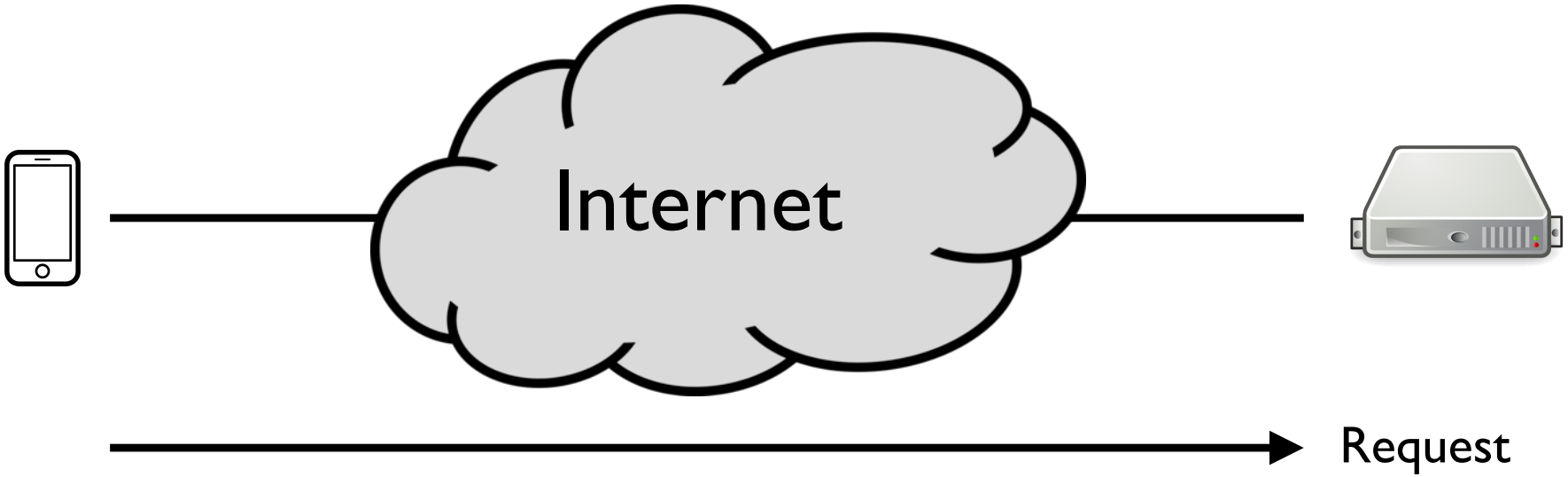
Application Layer	HTTP	Application-level information and data transfer: e.g., request and reply for document.
Transport Layer	TCP	Delivers data between applications (multiple applications on a computer).
Network Layer	IP	Delivers data between computers connected across multiple hops (the Internet).
Link Layer	Ethernet	Delivers data between computers directly connected through a medium (wire/wireless): single hop

Wireshark Demo

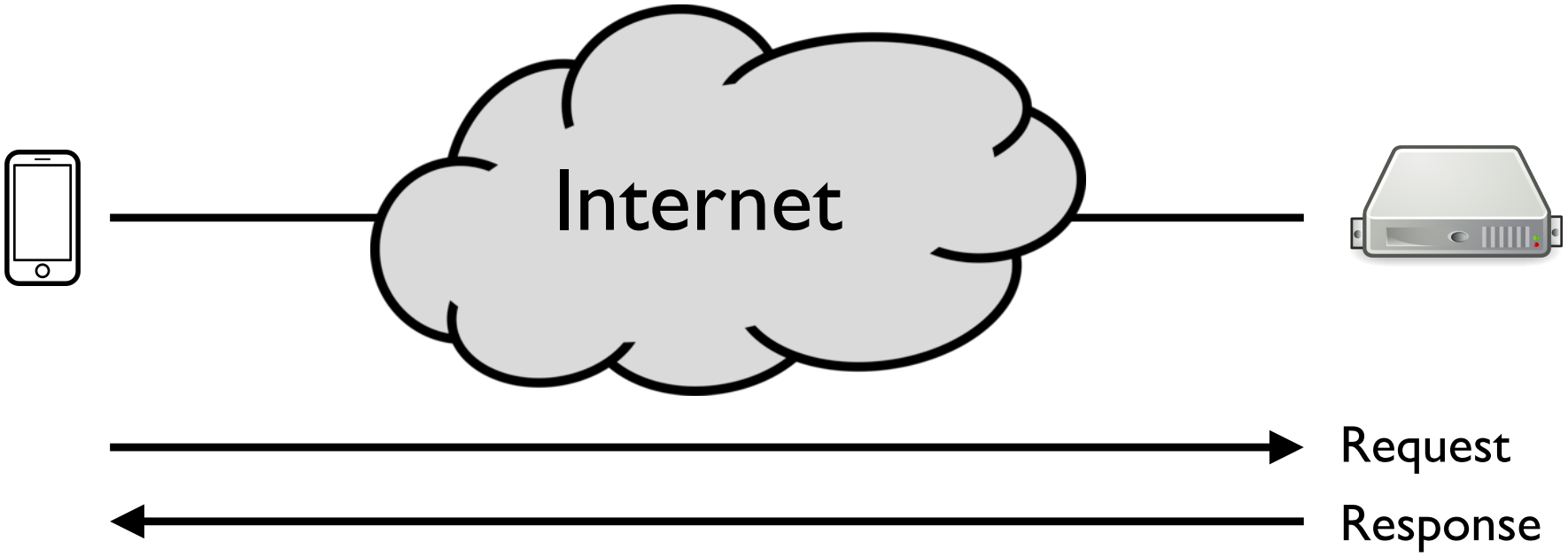
Conceptual Model of HTTP



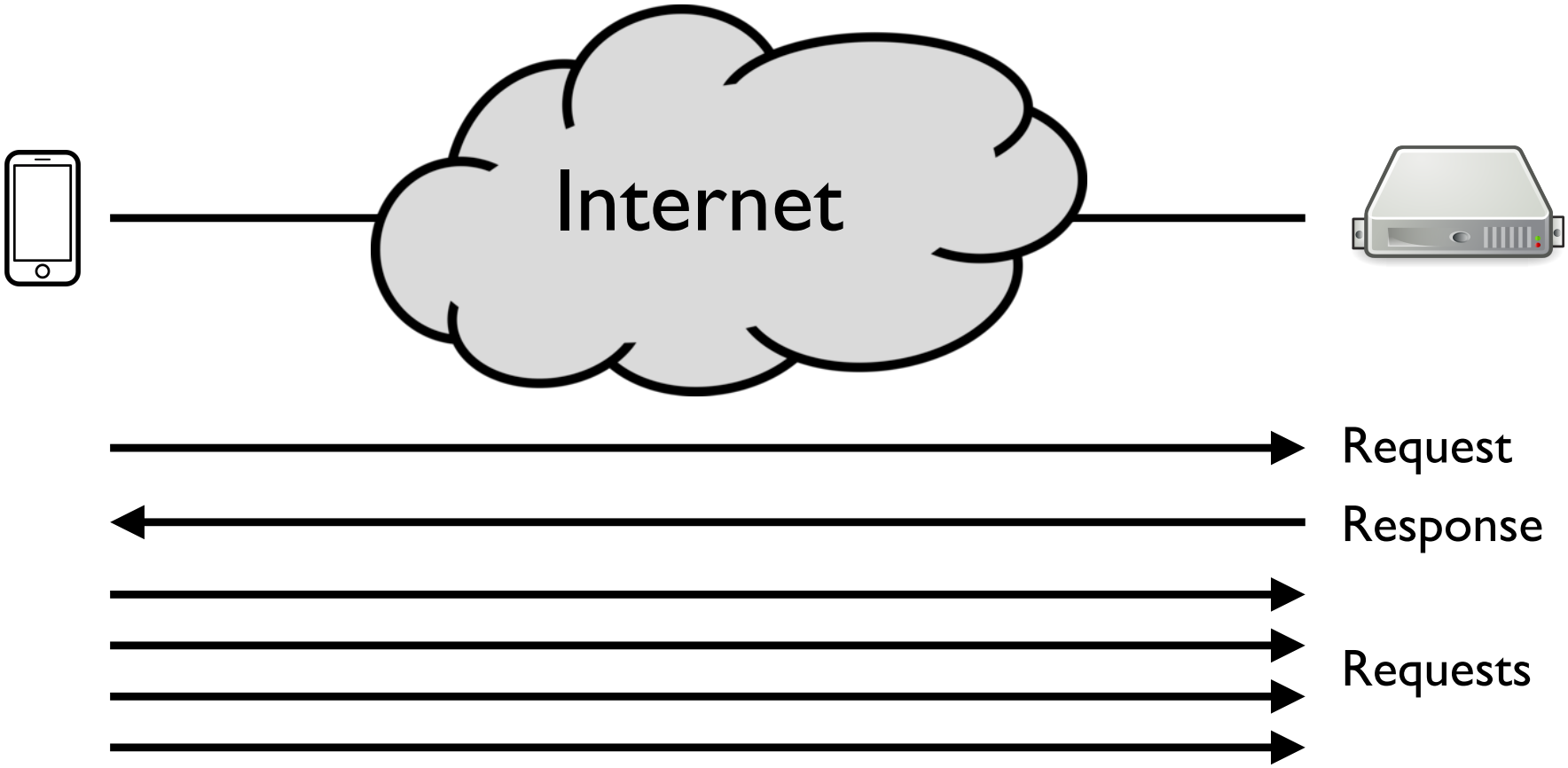
Conceptual Model of HTTP



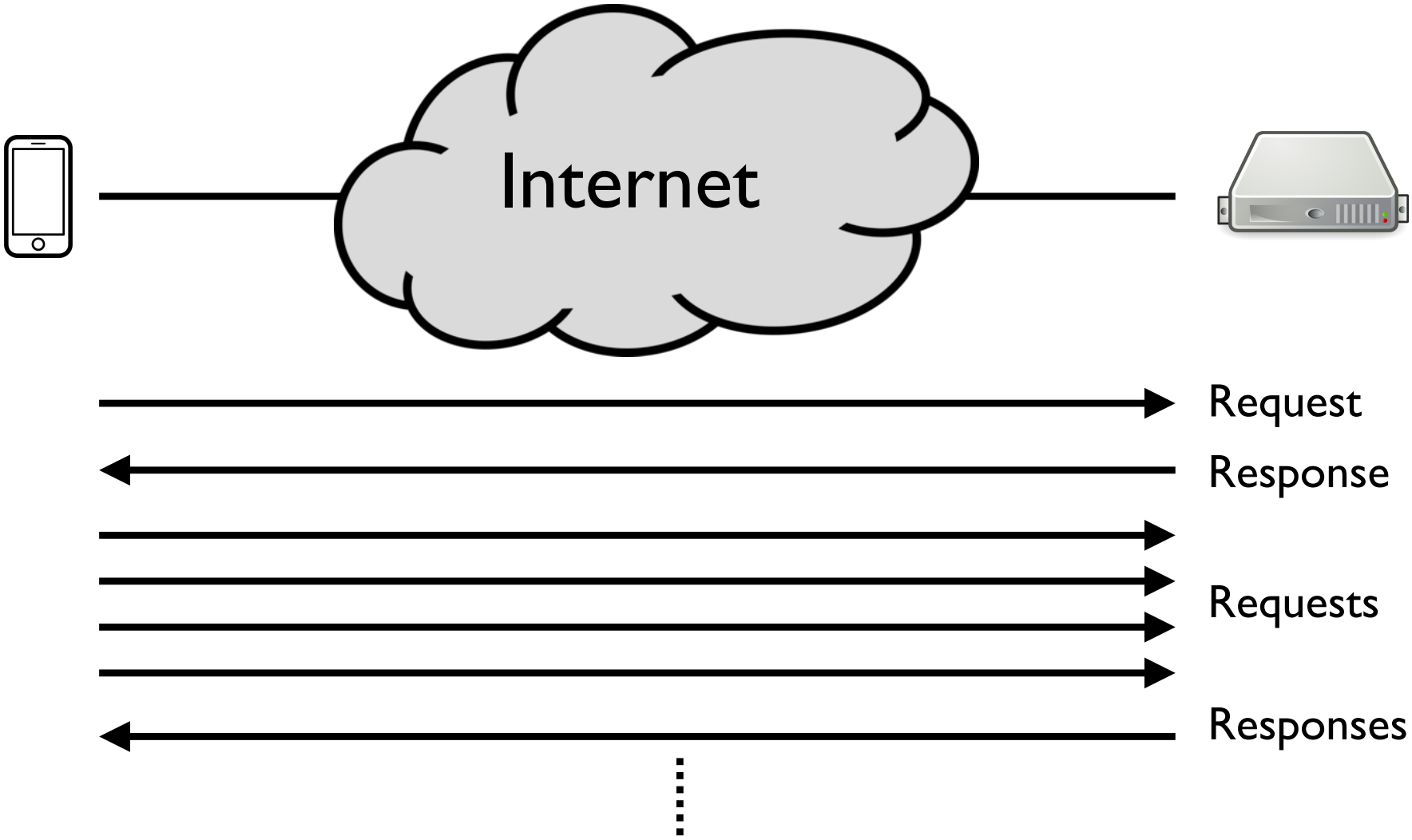
Conceptual Model of HTTP



Conceptual Model of HTTP

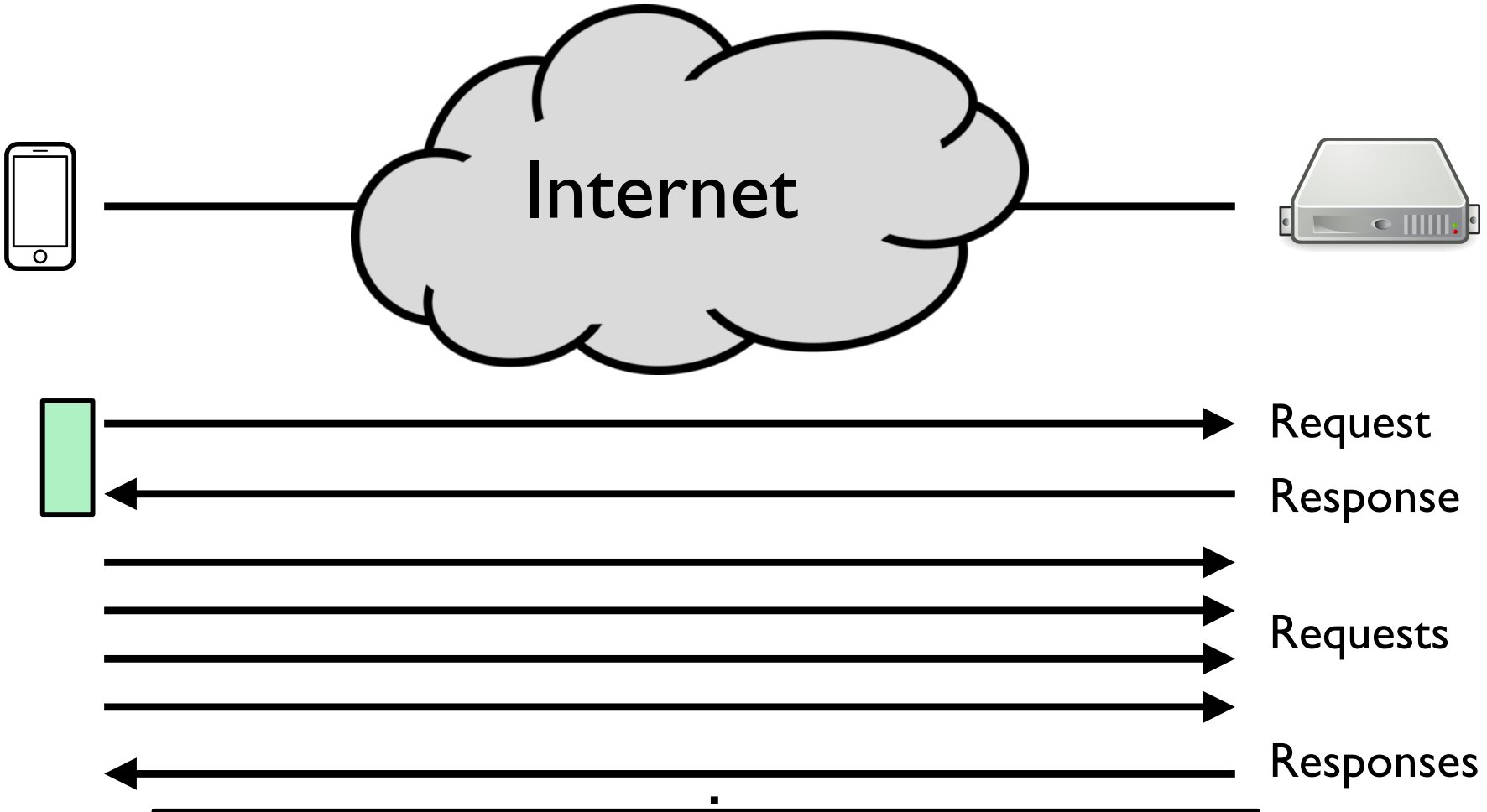


Conceptual Model of HTTP



Web Developer Demo

Conceptual Model of HTTP

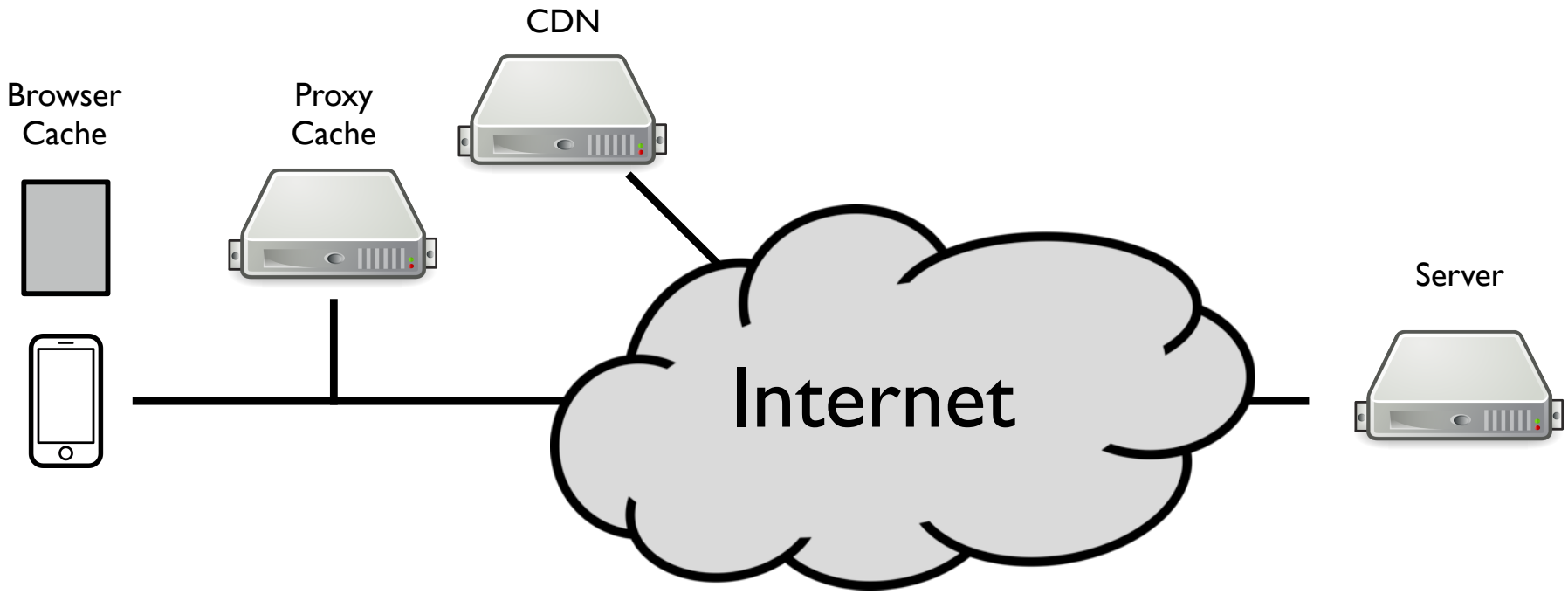


Page loading times greatly affected by latency: how long between a client sending a request and receiving the response.

Making HTTP faster

Basic answer: caching

- Bring data closer to client
- Browser cache, proxy caches, content distribution networks (CDNs)

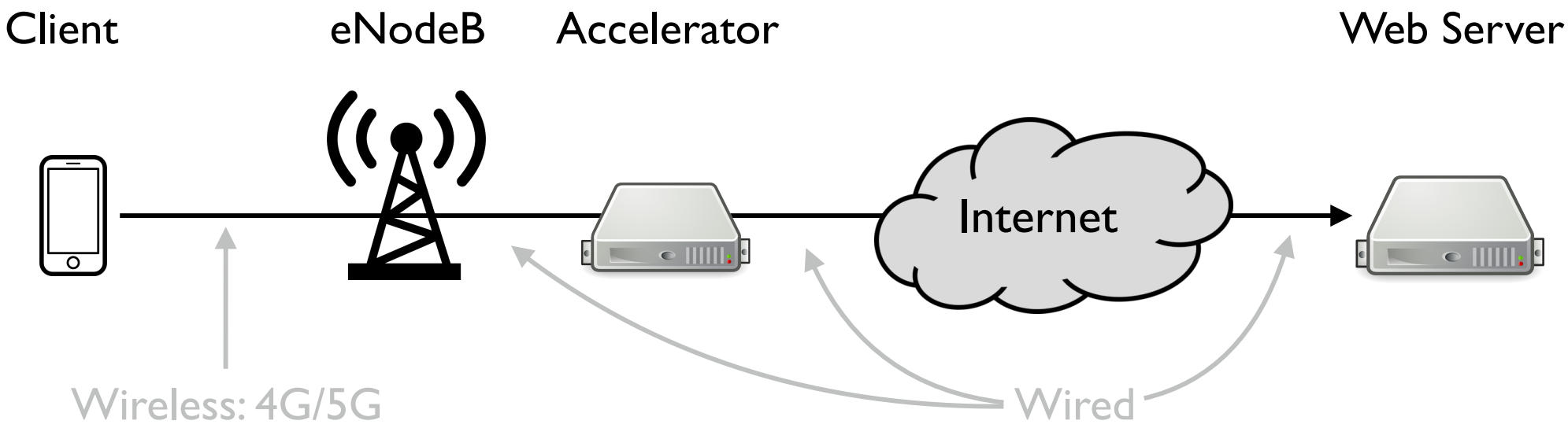


Caches vs. Accelerators

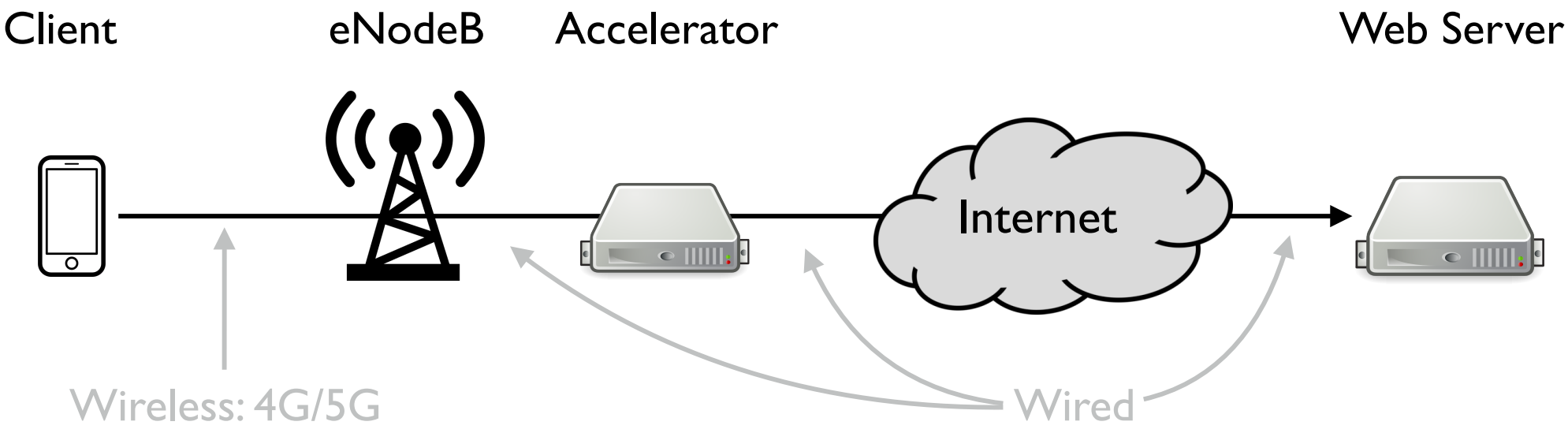
Caches store copies of previously accessed files

Accelerators take a more active role: *prefetch* files before the client requests them

Under the Hood: Mobile Web Access



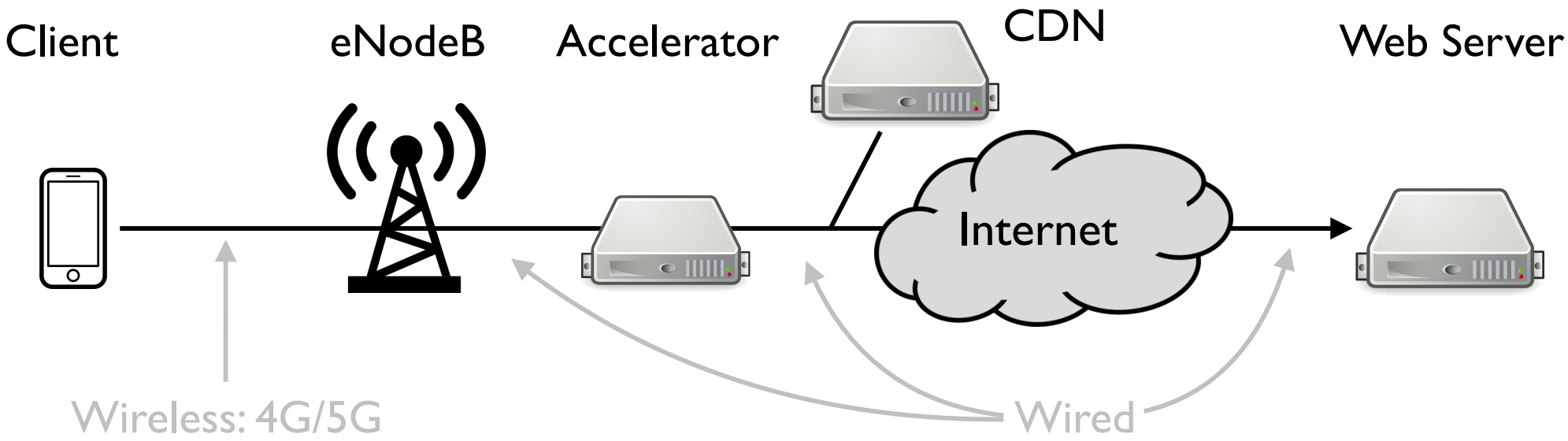
Mobile Web Access



What is the end-to-end latency of this communication path?

How is that latency distributed?

Mobile Web Access



What is the end-to-end latency of this communication path?

How is that latency distributed?

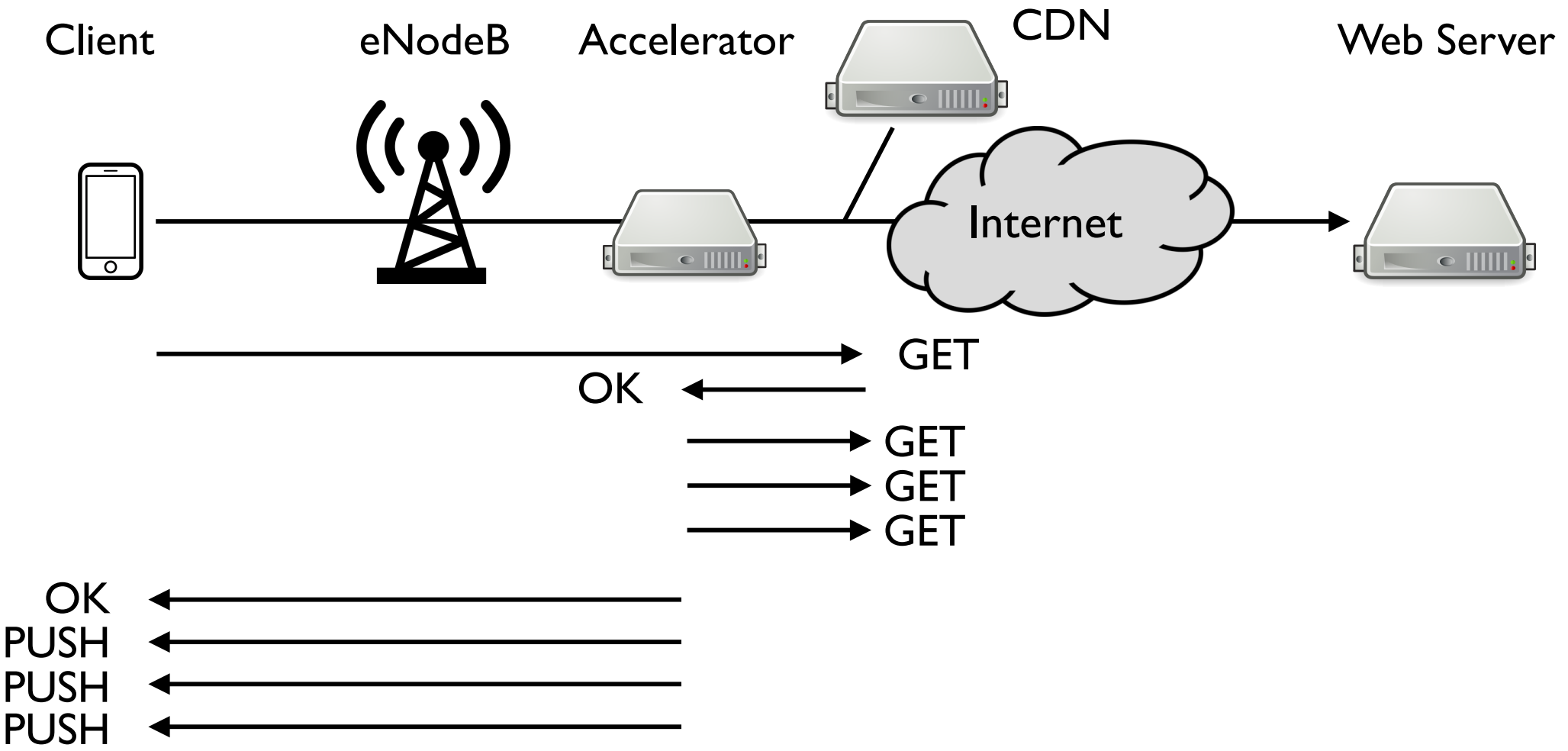
CDN nodes may be co-located in mobile operator network: latency can shift from being most Internet to being mostly wireless/last mile

Page Accelerators

Accelerators can observe your HTTP responses, see what files you're going to need, and prefetch them before you receive your response.

HTTP/2.0 allows server to PUSH data to a client before it requests it.

Mobile Web Access



HTTP is Cleartext

HTTP is a text-based protocol

- HTTP/2.0 introduces a bunch of forms of compression for mobile links, but it's still basically text

If you access an HTTP site, anyone can see your requests and the resulting data

- Can see what you request
- Can see the responses

HTTPS adds transport-layer security (TLS)

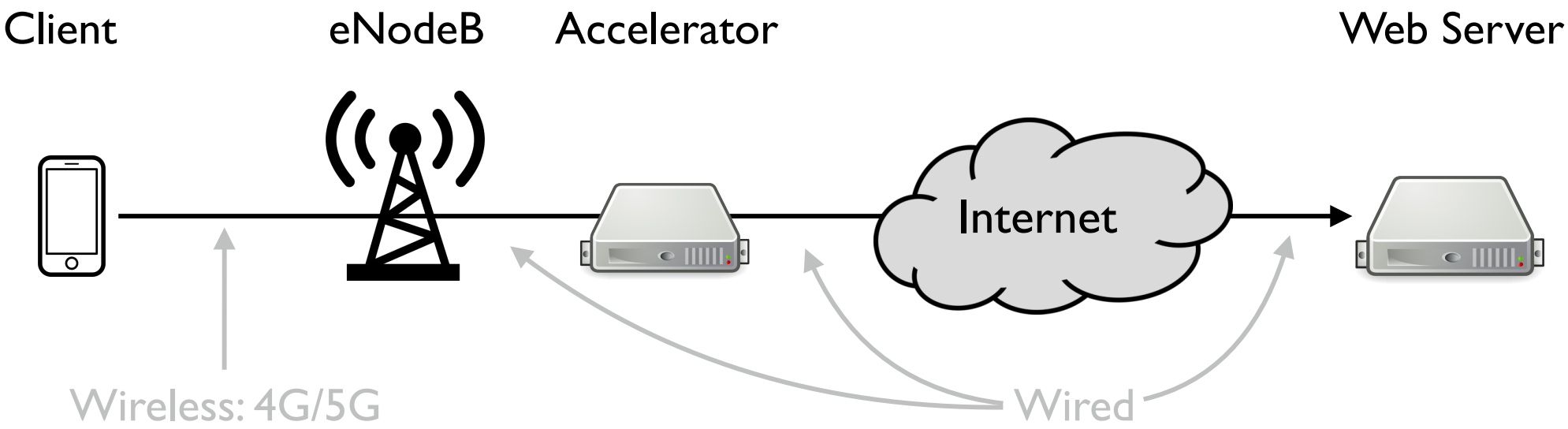
Networking Stack

Application Layer	HTTP	Application-level information and data transfer: e.g., request and reply for document.
Transport Layer	TCP	Delivers data between applications (multiple applications on a computer).
Network Layer	IP	Delivers data between computers connected across multiple hops (the Internet).
Link Layer	Ethernet	Delivers data between computers directly connected through a medium (wire/wireless): single hop

Networking Stack

Application Layer	HTTPS	Application-level information and data transfer: e.g., request and reply for document.
Session Layer	TLS	End-to-end confidentiality and integrity.
Transport Layer	TCP	Delivers data between applications (multiple applications on a computer).
Network Layer	IP	Delivers data between computers connected across multiple hops (the Internet).
Link Layer	Ethernet	Delivers data between computers directly connected through a medium (wire/wireless): single hop

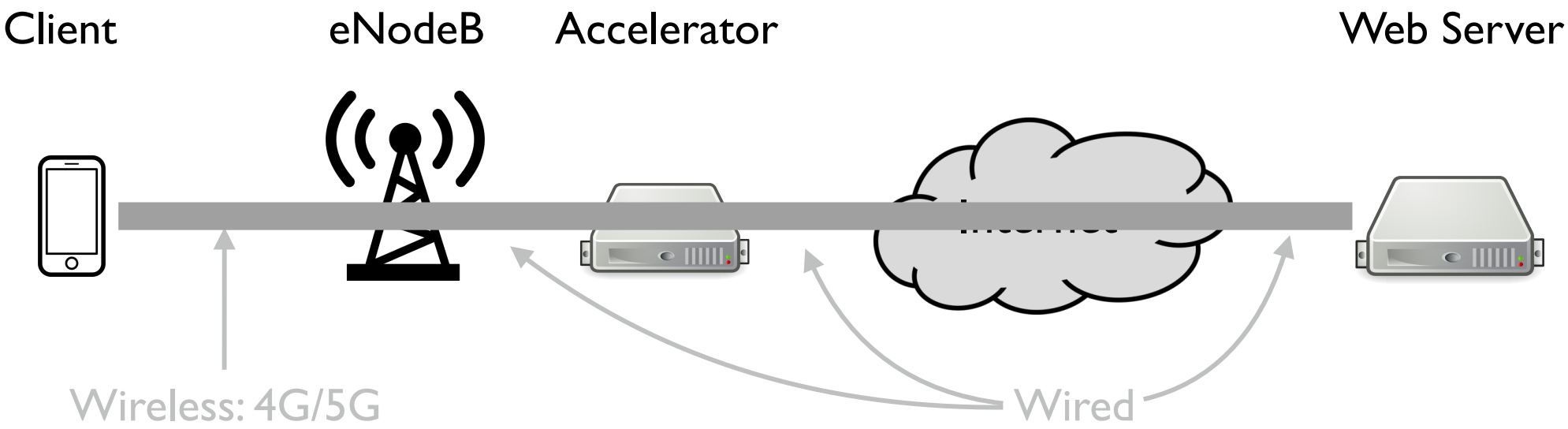
Mobile Web Access



What is the end-to-end latency of this communication path?

How is that latency distributed?

Mobile Web Access with HTTPS



With TLS, the data stream between the client and web server has both *confidentiality* and *integrity*

Accelerators can neither see nor change the data: they can't prefetch because they can't see the pages loaded

Symbolic Execution

Symbolic execution is a way of executing a program to see how inputs affect control flow and behavior

Variables are either *symbolic* (consider all possible values) or *concrete* (has a particular value)

When you encounter an operation on a symbolic value, consider what might happen

Symbolic execution example

```
int main(int argc, char** argv) {
  if (argc > 5) {

  }
  if (argc == 4) {
    char ch = *argv[argc];
  } ...
}
```

argc = *

argc > 5

argc = 4

possibly detect bug

Make argc and argv symbolic

- Consider what might happen for any possible input
- Can figure out two clauses are mutually exclusive

Constraints rapidly explode: need fast constraint solvers to figure out what might execute

KLEE (OSDI 2008)

KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs

Cristian Cadar, Daniel Dunbar, Dawson Engler *
Stanford University

One of 2 papers I know of that won both best paper *and* test of time

Abstract

We present a new symbolic execution tool, KLEE, capable of automatically generating tests that achieve high coverage on a diverse set of complex and environmentally-intensive programs. We used KLEE to thoroughly check all 89 stand-alone programs in the GNU COREUTILS utility suite, which form the core user-level environment installed on millions of Unix systems, and arguably are the single most heavily tested set of open-source programs in existence. KLEE-generated tests achieve high line coverage — on average over 90% per tool (median: over 94%) — and significantly beat the coverage of the developers' own hand-written test suite. When we did the same for 75 equivalent tools in the BUSYBOX embedded system suite, results were even better, including 100% coverage on 31 of them.

We also used KLEE as a bug finding tool, applying it to 452 applications (over 430K total lines of code), where it found 56 serious bugs, including three in COREUTILS that had been missed for over 15 years. Finally, we used

bolic values and replace corresponding concrete program operations with ones that manipulate symbolic values. When program execution branches based on a symbolic value, the system (conceptually) follows both branches, on each path maintaining a set of constraints called the *path condition* which must hold on execution of that path. When a path terminates or hits a bug, a test case can be generated by solving the current path condition for concrete values. Assuming deterministic code, feeding this concrete input to a raw, unmodified version of the checked code will make it follow the same path and hit the same bug.

Results are promising. However, while researchers have shown such tools can sometimes get good coverage and find bugs on a small number of programs, it has been an open question whether the approach has any hope of consistently achieving high coverage on real applications. Two common concerns are (1) the exponential number of paths through code and (2) the challenges in handling code that interacts with its surrounding environment, such as the operating system, the network, or

```
1 : void expand(char *arg, unsigned char *buffer) {          8
2 :   int i, ac;                                             9
3 :   while (*arg) {                                        10*
4 :     if (*arg == '\\') {                                11*
5 :       arg++;
6 :       i = ac = 0;
7 :       if (*arg >= '0' && *arg <= '7') {
8 :         do {
9 :           ac = (ac << 3) + *arg++ - '0';
10:           i++;
11:         } while (i<4 && *arg>='0' && *arg<='7');
12:         *buffer++ = ac;
13:       } else if (*arg != '\0')
14:         *buffer++ = *arg++;
15:     } else if (*arg == '[') {                            12*
16:       arg++;                                             13
17:       i = *arg++;                                        14
18:       if (*arg++ != '-') {                               15!
19:         *buffer++ = '[';
20:         arg --= 2;
21:         continue;
22:       }
23:       ac = *arg++;
24:       while (i <= ac) *buffer++ = i++;
25:       arg++;       /* Skip ']' */
26:     } else
27:       *buffer++ = *arg++;
28:   }
29: }
30: ...
31: int main(int argc, char* argv[]) {                        1
32:   int index = 1;                                         2
33:   if (argc > 1 && argv[index][0] == '-') {               3*
34:     ...                                                  4
35:   }                                                      5
36:   ...                                                    6
37:   expand(argv[index++], index);                          7
38:   ...
39: }
```

case (`tr [ "" ""`) that hits it. Assuming the options of the previous subsection, KLEE runs `tr` as follows:

1 KLEE constructs symbolic command line string arguments whose contents have no constraints other than zero-termination. It then constrains the number of arguments to be between 0 and 3, and their sizes to be 1, 10 and 10 respectively. It then calls `main` with these initial path constraints.

2 When KLEE hits the branch `argc > 1` at line 33, it uses its constraint solver STP [23] to see which directions can execute given the current path condition. For this branch, both directions are possible; KLEE forks execution and follows both paths, adding the constraint `argc` > 1 on the false path and `argc` ≤ 1 on the true path.

3 Given more than one active path, KLEE must pick which one to execute first. We describe its algorithm in Section 3.4. For now assume it follows the path that reaches the bug. As it does so, KLEE adds further constraints to the contents of `arg`, and forks for a total of five times (lines denoted with a "`*`"): twice on line 33, and then on lines 3, 4, and 15 in `expand`.

4 At each dangerous operation (e.g., pointer dereference), KLEE checks if any possible value allowed by the current path condition would cause an error. On the annotated path, KLEE detects no errors before line 18. At that point, however, it determines that input values exist that allow the read of `arg` to go out of bounds: after taking the true branch at line 15, the code increments `arg` twice without checking if the string has ended. If it has, this increment skips the terminating `'\0'` and points to invalid memory.

5 KLEE generates concrete values for `argc` and `argv` (i.e., `tr [ "" ""`) that when rerun on a raw version of `tr` will hit this bug. It then continues following the current path, adding the constraint that the error does not occur (in order to find other errors).

```
1 : void expand(char *arg, unsigned char *buffer) {          8
2 :   int i, ac;                                             9
3 :   while (*arg) {                                        10*
4 :     if (*arg == '\\') {                                11*
5 :       arg++;
6 :       i = ac = 0;
7 :       if (*arg >= '0' && *arg <= '7') {
8 :         do {
9 :           ac = (ac << 3) + *arg++ - '0';
10:           i++;
11:         } while (i<4 && *arg>='0' && *arg<='7');
12:         *buffer++ = ac;
13:       } else if (*arg != '\0')
14:         *buffer++ = *arg++;
15:     } else if (*arg == '[') {                            12*
16:       arg++;                                             13
17:       i = *arg++;                                        14
18:       if (*arg++ != '-') {                               15!
19:         *buffer++ = '[';
20:         arg --= 2;
21:         continue;
22:       }
23:       ac = *arg++;
24:       while (i <= ac) *buffer++ = i++;
25:       arg++;       /* Skip ']' */
26:     } else
27:       *buffer++ = *arg++;
28:   }
29: }
30: ...
31: int main(int argc, char* argv[]) {                        1
32:   int index = 1;                                         2
33:   if (argc > 1 && argv[index][0] == '-') {               3*
34:     ...                                                  4
35:   }                                                      5
36:   ...                                                    6
37:   expand(argv[index++], index);                          7
38:   ...
39: }
```

case (`tr [ "" ""`) that hits it. Assuming the options of the previous subsection, KLEE runs `tr` as follows:

1 KLEE constructs symbolic command line string arguments whose contents have no constraints other than zero-termination. It then constrains the number of arguments to be between 0 and 3, and their sizes to be 1, 10 and 10 respectively. It then calls `main` with these initial path constraints.

2 When KLEE hits the branch `argc > 1` at line 33, it uses its constraint solver STP [23] to see which directions can execute given the current path condition. For this branch, both directions are possible; KLEE forks execution and follows both paths, adding the constraint `argc` > 1 on the false path and `argc` ≤ 1 on the true path.

3 Given more than one active path, KLEE must pick which one to execute first. We describe its algorithm in Section 3.4. For now assume it follows the path that reaches the bug. As it does so, KLEE adds further constraints to the contents of `arg`, and forks for a total of five times (lines denoted with a "`*`"): twice on line 33, and then on lines 3, 4, and 15 in `expand`.

4 At each dangerous operation (e.g., pointer dereference), KLEE checks if any possible value allowed by the current path condition would cause an error. On the annotated path, KLEE detects no errors before line 18. At that point, however, it determines that input values exist that allow the read of `arg` to go out of bounds: after taking the true branch at line 15, the code increments `arg` twice without checking if the string has ended. If it has, this increment skips the terminating `'\0'` and points to invalid memory.

5 KLEE generates concrete values for `argc` and `argv` (i.e., `tr [ "" ""`) that when rerun on a raw version of `tr` will hit this bug. It then continues following the current path, adding the constraint that the error does not occur (in order to find other errors).

```
1 : void expand(char *arg, unsigned char *buffer) {          8
2 :   int i, ac;                                             9
3 :   while (*arg) {                                        10*
4 :     if (*arg == '\\') {                                11*
5 :       arg++;
6 :       i = ac = 0;
7 :       if (*arg >= '0' && *arg <= '7') {
8 :         do {
9 :           ac = (ac << 3) + *arg++ - '0';
10:           i++;
11:         } while (i<4 && *arg>='0' && *arg<='7');
12:         *buffer++ = ac;
13:       } else if (*arg != '\0')
14:         *buffer++ = *arg++;
15:     } else if (*arg == '[') {                            12*
16:       arg++;                                             13
17:       i = *arg++;                                        14
18:       if (*arg++ != '-') {                               15!
19:         *buffer++ = '[';
20:         arg --= 2;
21:         continue;
22:       }
23:       ac = *arg++;
24:       while (i <= ac) *buffer++ = i++;
25:       arg++;       /* Skip ']' */
26:     } else
27:       *buffer++ = *arg++;
28:   }
29: }
30: ...
31: int main(int argc, char* argv[]) {                        1
32:   int index = 1;                                         2
33:   if (argc > 1 && argv[index][0] == '-') {               3*
34:     ...                                                  4
35:   }                                                      5
36:   ...                                                    6
37:   expand(argv[index++], index);                          7
38:   ...
39: }
```

case (`tr [ "" ""`) that hits it. Assuming the options of the previous subsection, KLEE runs `tr` as follows:

1 KLEE constructs symbolic command line string arguments whose contents have no constraints other than zero-termination. It then constrains the number of arguments to be between 0 and 3, and their sizes to be 1, 10 and 10 respectively. It then calls `main` with these initial path constraints.

2 When KLEE hits the branch `argc > 1` at line 33, it uses its constraint solver STP [23] to see which directions can execute given the current path condition. For this branch, both directions are possible; KLEE forks execution and follows both paths, adding the constraint `argc` > 1 on the false path and `argc` ≤ 1 on the true path.

3 Given more than one active path, KLEE must pick which one to execute first. We describe its algorithm in Section 3.4. For now assume it follows the path that reaches the bug. As it does so, KLEE adds further constraints to the contents of `arg`, and forks for a total of five times (lines denoted with a "`*`"): twice on line 33, and then on lines 3, 4, and 15 in `expand`.

4 At each dangerous operation (e.g., pointer dereference), KLEE checks if any possible value allowed by the current path condition would cause an error. On the annotated path, KLEE detects no errors before line 18. At that point, however, it determines that input values exist that allow the read of `arg` to go out of bounds: after taking the true branch at line 15, the code increments `arg` twice without checking if the string has ended. If it has, this increment skips the terminating `'\0'` and points to invalid memory.

5 KLEE generates concrete values for `argc` and `argv` (i.e., `tr [ "" ""`) that when rerun on a raw version of `tr` will hit this bug. It then continues following the current path, adding the constraint that the error does not occur (in order to find other errors).

```
1 : void expand(char *arg, unsigned char *buffer) {          8
2 :   int i, ac;                                             9
3 :   while (*arg) {                                        10*
4 :     if (*arg == '\\') {                                11*
5 :       arg++;
6 :       i = ac = 0;
7 :       if (*arg >= '0' && *arg <= '7') {
8 :         do {
9 :           ac = (ac << 3) + *arg++ - '0';
10:           i++;
11:         } while (i<4 && *arg>='0' && *arg<='7');
12:         *buffer++ = ac;
13:       } else if (*arg != '\0')
14:         *buffer++ = *arg++;
15:     } else if (*arg == '[') {                            12*
16:       arg++;                                             13
17:       i = *arg++;                                        14
18:       if (*arg++ != '-') {                               15!
19:         *buffer++ = '[';
20:         arg --= 2;
21:         continue;
22:       }
23:       ac = *arg++;
24:       while (i <= ac) *buffer++ = i++;
25:       arg++;       /* Skip ']' */
26:     } else
27:       *buffer++ = *arg++;
28:   }
29: }
30: ...
31: int main(int argc, char* argv[]) {                        1
32:   int index = 1;                                         2
33:   if (argc > 1 && argv[index][0] == '-') {               3*
34:     ...                                                  4
35:   }                                                      5
36:   ...                                                    6
37:   expand(argv[index++], index);                          7
38:   ...
39: }
```

case (`tr [ "" ""`) that hits it. Assuming the options of the previous subsection, KLEE runs `tr` as follows:

1 KLEE constructs symbolic command line string arguments whose contents have no constraints other than zero-termination. It then constrains the number of arguments to be between 0 and 3, and their sizes to be 1, 10 and 10 respectively. It then calls `main` with these initial path constraints.

2 When KLEE hits the branch `argc > 1` at line 33, it uses its constraint solver STP [23] to see which directions can execute given the current path condition. For this branch, both directions are possible; KLEE forks execution and follows both paths, adding the constraint `argc` > 1 on the false path and `argc` ≤ 1 on the true path.

3 Given more than one active path, KLEE must pick which one to execute first. We describe its algorithm in Section 3.4. For now assume it follows the path that reaches the bug. As it does so, KLEE adds further constraints to the contents of `arg`, and forks for a total of five times (lines denoted with a "*"): twice on line 33, and then on lines 3, 4, and 15 in `expand`.

4 At each dangerous operation (e.g., pointer dereference), KLEE checks if any possible value allowed by the current path condition would cause an error. On the annotated path, KLEE detects no errors before line 18. At that point, however, it determines that input values exist that allow the read of `arg` to go out of bounds: after taking the true branch at line 15, the code increments `arg` twice without checking if the string has ended. If it has, this increment skips the terminating `'\0'` and points to invalid memory.

5 KLEE generates concrete values for `argc` and `argv` (i.e., `tr [ "" ""`) that when rerun on a raw version of `tr` will hit this bug. It then continues following the current path, adding the constraint that the error does not occur (in order to find other errors).