Server-Side Secret Sauce

Rapidly Prototyping
a WebSocket Application

Danyil Bohdan

The Recipe

websocketd

\+ netcat

\+ Tcl

(or your favorite scripting language)

---

easy rapid prototyping

websocketd

- “WebSockets the UNIX way”
- Faster than many languages’ native WebSocket implementation [2]
- Low latency overhead (3 ms roundtrip)
- Neat additional features
  - Static file server (HTTP server)
  - HTTPS
  - CGI
  - Can route different URLs to different programs

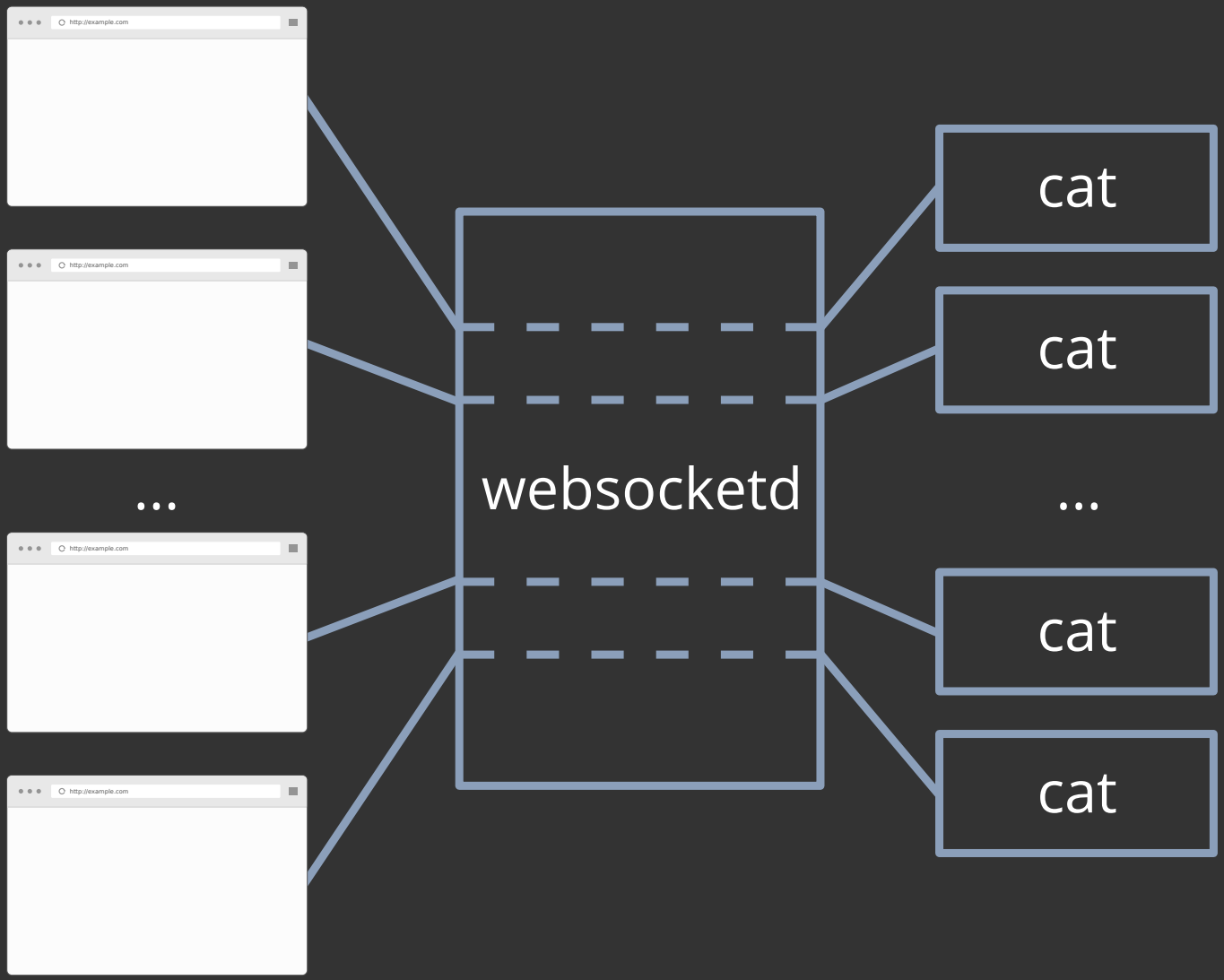
“Party like it’s 1999!”

- CGI (as in `/cgi-bin/`, not as in Michael Bay) is back.
- Wasn't CGI slow and bad?
- Not slow per se but slow and expensive *to start*.
- Doesn't matter because our processes are long-lived.

“Party like it’s 1999!”

- CGI (as in `/cgi-bin/`, not as in Michael Bay) is back.
- Wasn’t CGI slow and bad?
- Not slow per se but slow and expensive to start.
- Doesn’t matter because our processes are long-lived.

Example 1 — Chart



Example 1 — Client

```
<!DOCTYPE html>
<html>
<head>
    <title>Echo</title>
</head>
<body>
    <script type="text/javascript">
        var ws = new WebSocket("ws://localhost:8080/", "ex1");
        ws.onmessage = function(event) {
            window.alert(event.data);
        };
        ws.onopen = function(event) {
            ws.send('Hello!');
        };
    </script>
</body>
</html>
```

Example 1 — Server

```
#!/bin/sh
# 'cat' prints its input to standard output.
../websocketd --port=8080 --staticdir=. cat
```

Example 1 — Data flow

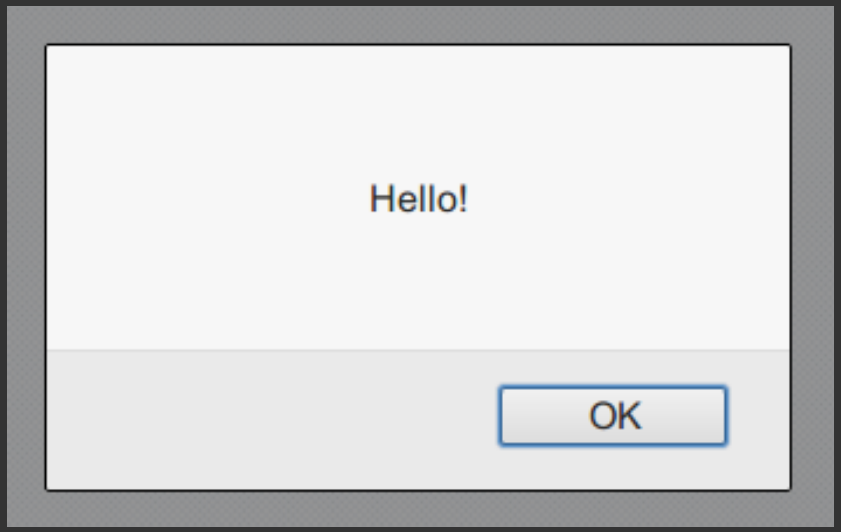
Example 1 — Data flow

Example 1 — Data flow

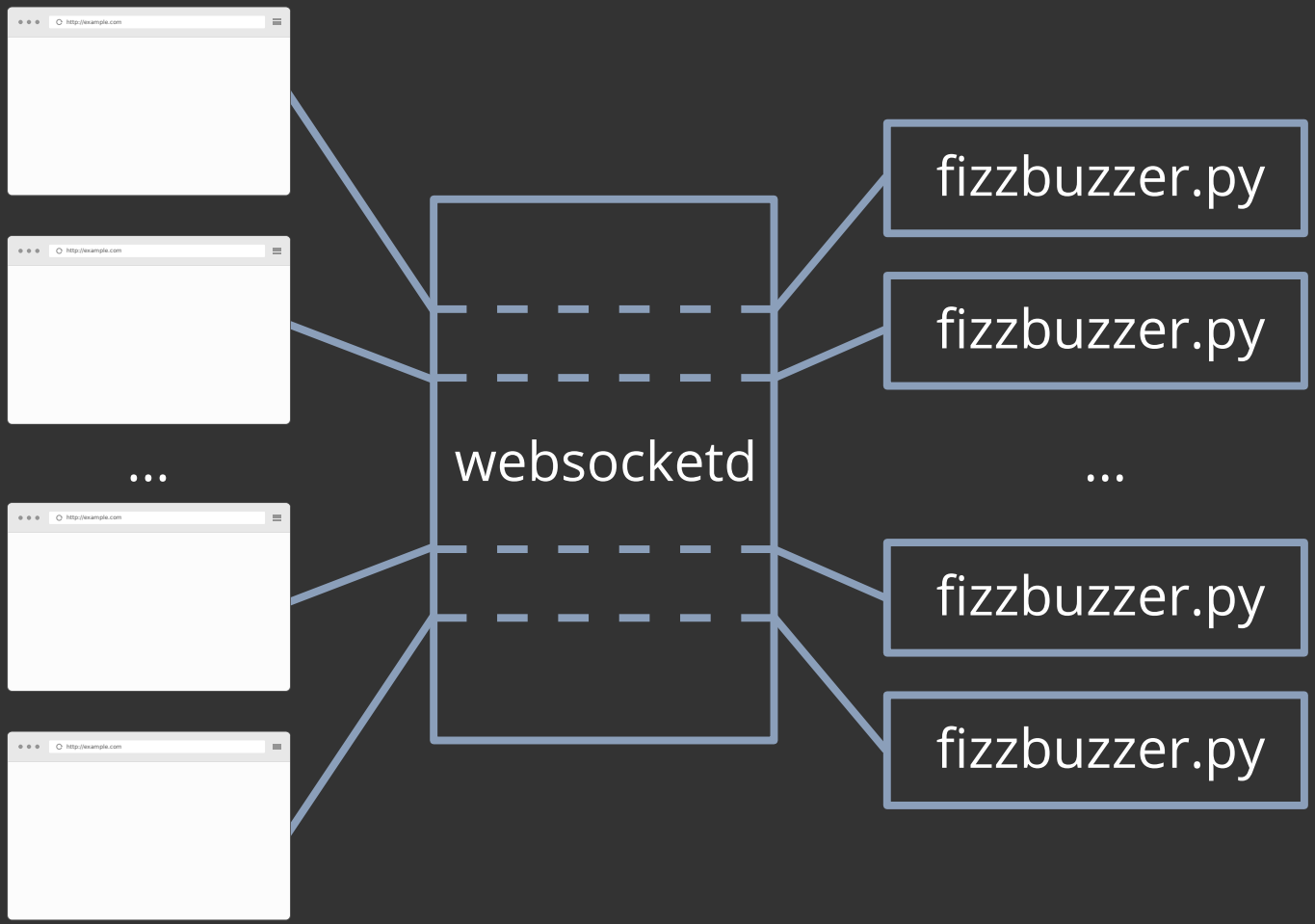
Example 1 — Data flow

Example 1 — Data flow

Example 1 — Screenshot



Example 2 — Chart



Example 2 — Client

```
<!DOCTYPE html>
<html>
<head>
    <title>FaaS – FizzBuzz as a Service</title>
</head>
<body>
    <script type="text/javascript">
        var ws = new WebSocket("ws://localhost:8081/", "ex2");
        ws.onmessage = function(event) {
            console.log(event.data);
        };
        ws.onopen = function(event) {
            for (var i = 0; i <= 100; i++) {
                ws.send(i);
            };
        };
    </script>
</body>
</html>
```

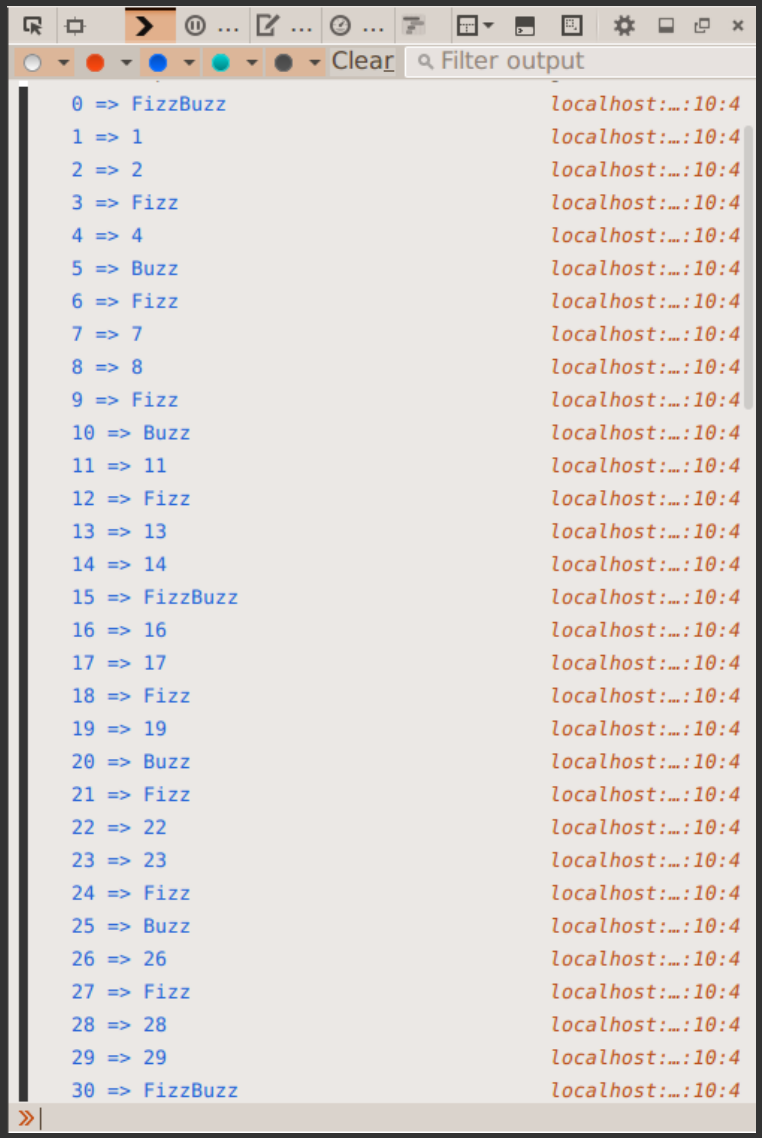
Example 2 — Server (1)

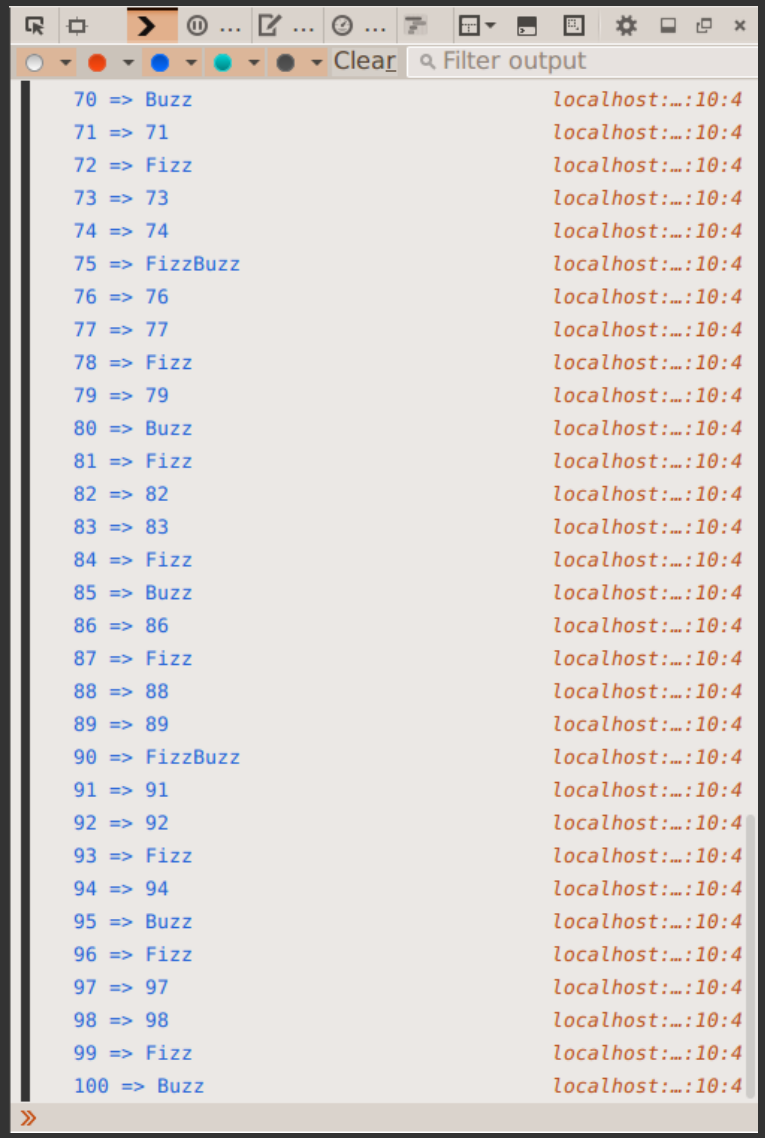
```
#!/bin/sh
../websocketd --port=8081 --staticdir=. ./fizzbuzzer.py
```

Example 2 — Server (2)

```
#!/usr/bin/env python3
def to_fizzbuzz(n):
    if n % 15 == 0:
        return "FizzBuzz"
    elif n % 3 == 0:
        return "Fizz"
    elif n % 5 == 0:
        return "Buzz"
    else:
        return n

while True:
    try:
        n = int(input()) # Synchronous. We wait for input.
        print("{0} => {1}".format(n, to_fizzbuzz(n)))
    except EOFError:
        break
    except ValueError:
        print("error: cannot process input")
```

Example 2 — Screenshot





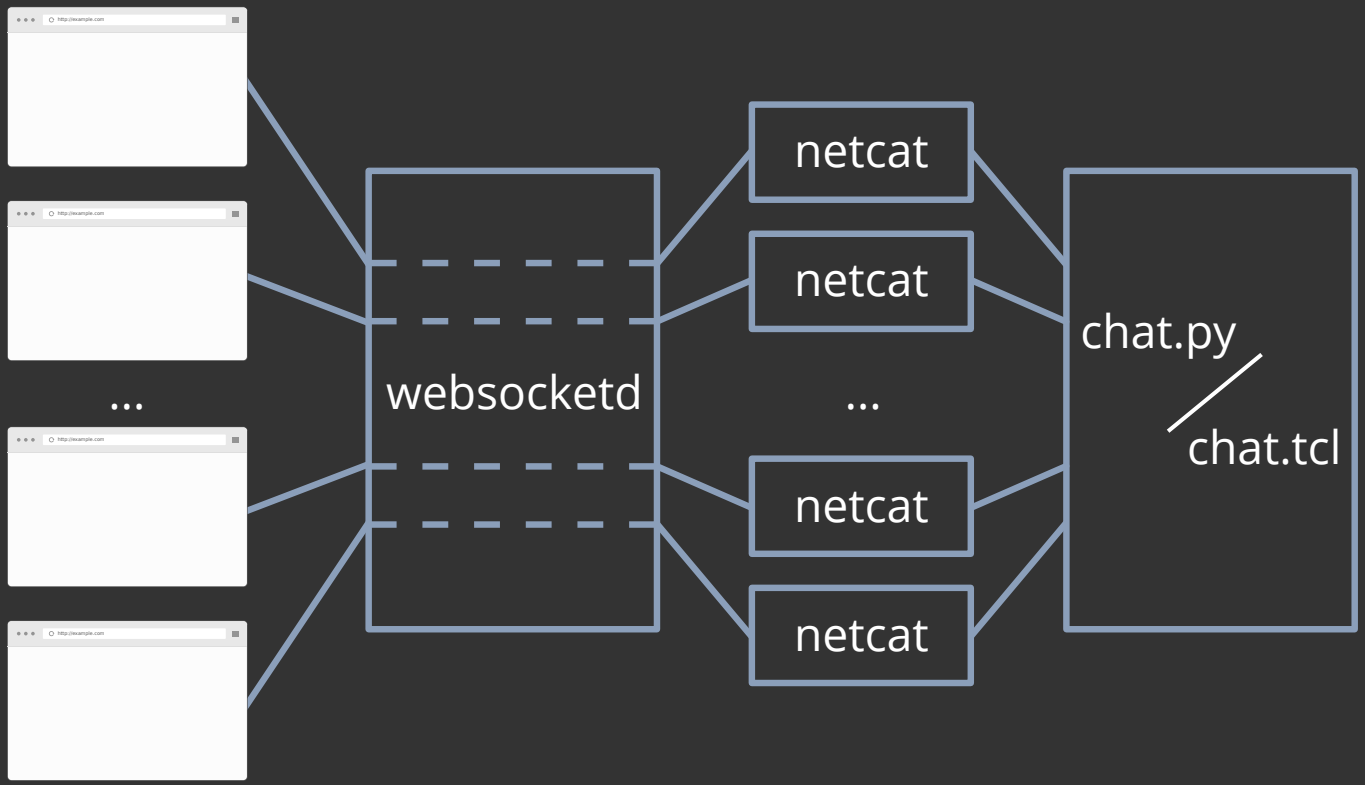
netcat

- “The TCP/IP Swiss Army knife”
- A universal socket client/server
- Talk to sockets from the command line
  - `printf "GET / HTTP/1.1\r\nHost: example.com\r\n\r\n" | nc example.com 80`
- Probably already there on your Linux, FreeBSD or OS X system, available [3] for Windows

Plumbing with netcat

- websocketd spawns a separate process for each connection
  - What if you wanted the clients to interact with each other?
- Have a TCP server listen on a regular TCP socket
- Redirect WebSocket connections to it
  - `./websocketd-v0.2.11-linux-x86_64 --port=8080 nc localhost 7777`

Example 3 — Chart



Example 3 — Client (1)

```
<!DOCTYPE html>
<html>
<head>
    <style type="text/css">
        #chat {
            width: 700px; height: 500px; overflow: scroll;
        }
    </style>
</head>
<body>
    <div id="chat"></div><br>
    <input type="text" id="input">
    <input type="button" id="send" value="send">

    <script type="text/javascript" src="client.js"></script>
</body>
</html>
```

Example 3 — Client (2)

```
// client.js
var qs = document.querySelector;
var inputElem = qs('#input');
var chatAreaElem = qs('#chat');
var sendElem = qs('#send');
var post_chat_message = function(text) {
    // ...
};

var ws = new WebSocket("ws://localhost:8082/", "ex3");
ws.onopen = function(event) {
    var sendFunc = function() {
        ws.send(input.value); input.value = '';
    }
    sendElem.addEventListener('click', sendFunc);
    inputElem.addEventListener('keypress', function(event) {
        var key = event.which || event.keyCode;
        if (key === 13) { sendFunc(); }
    });
}
ws.onmessage = function(event) { post_chat_message(event.data); };
ws.onclose = function(event) {
    post_chat_message('Connection closed.')
}
```

Example 3 — Server (1)

```
#!/bin/sh
set -e

# Run chat.py or chat.tcl depending on the command line arg.
if [ "$1" = "py" ] || [ "$1" = "tcl" ]; then
    echo "Starting chat.$1"
    ./chat.$1 &
else
    echo "usage: $0 (py|tcl)"
    exit 1
fi

# Kill the chat server on exit.
trap 'kill $(jobs -pr)' SIGINT SIGTERM EXIT

# Redirect each WebSocket connection to localhost TCP socket.
../websocketd --port=8082 --staticdir=. nc localhost 7777
```

Example 3 — Server (2)

Python 3

```
#!/usr/bin/env python3
import asyncore
import socket

class ChatHandler(asyncore.dispatcher_with_send):
    def __init__(self, sock, name, server):
        asyncore.dispatcher_with_send.__init__(self, sock)
        self.name = name
        self.server = server

    def handle_read(self):
        data = self.recv(8192).decode()
        if data != "":
            self.server.broadcast(self.name + ": " + data)

    def handle_close(self):
        del self.server.clients[self.name]
        self.close()
        self.server.broadcast(self.name + " disconnected")

class ChatServer(asyncore.dispatcher):
    def __init__(self, host, port):
        asyncore.dispatcher.__init__(self)
        self.create_socket(socket.AF_INET, socket.SOCK_STREAM)
        self.set_reuse_addr()
        self.bind((host, port))
        self.listen(5)
        self.clients = {}
        self.counter = 0

    def broadcast(self, message):
        print(message)
        messageBytes = (message + "\n").encode()
        for otherName in self.clients:
            handler = self.clients[otherName]
            handler.send(messageBytes)

    def handle_accept(self):
        pair = self.accept()
        if pair is not None:
            sock, addr = pair
            self.counter += 1
            name = "client" + str(self.counter)
            print('Incoming connection from {0}: {1}'.
                  format(repr(addr), name))
            self.broadcast(name + " connected")
            handler = ChatHandler(sock, name, self)
            self.clients[name] = handler

server = ChatServer('localhost', 7777)
asyncore.loop()
```

Tcl

```
#!/usr/bin/env tclsh
namespace eval ::chat {
    variable count 0
    variable clients {}
}

proc ::chat::main {} {
    socket -server ::chat::new-connection-handler -myaddr 127.0.0.1 7777
    vwait forever
}

proc ::chat::new-connection-handler {channel clientAddr clientPort} {
    incr ::chat::count
    set name client$::chat::count
    lappend ::chat::clients $channel $name

    puts "Incoming connection from ('$clientAddr', $clientPort): $name"
    ::chat::broadcast "$name connected"

    fconfigure $channel -buffering line
    fileevent $channel readable [list ::chat::readable-handler $channel]
}

proc ::chat::broadcast message {
    puts $message
    foreach {otherChannel _} $::chat::clients {
        puts $otherChannel $message
    }
}

proc ::chat::readable-handler channel {
    set senderName [dict get $::chat::clients $channel]
    if {[gets $channel line] >= 0} {
        ::chat::broadcast "$senderName: $line"
    }
    if {[eof $channel]} {
        ::chat::broadcast "$senderName disconnected"
        close $channel
        dict unset ::chat::clients $channel
    }
}

::chat::main
```

Example 3 — Server (2)

Python 3

```
#!/usr/bin/env python3
import asyncore
import socket

class ChatHandler(asyncore.dispatcher_with_send):
    def __init__(self, sock, name, server):
        asyncore.dispatcher_with_send.__init__(self, sock)
        self.name = name
        self.server = server

    def handle_read(self):
        data = self.recv(8192).decode()
        if data != "":
            self.se      adcast(self              data,
          _close(
          f.serv      ts[se
            ()
            bro       self        disconne     ")
class      er          tcher
    def        __(s          t):
             e.dis          it__(self)
              eate_so        .AF_INET, socket.SOCK_STREAM)
        s         _reuse_a
        s          ((host, port))
        se        en(5)
        self.clients = {}
        self.counter = 0

    def broadcast(self, message):
        print(message)
        messageBytes = (message + "\n").encode()
        for otherName in self.clients:
            handler = self.clients[otherName]
            handler.send(messageBytes)

    def handle_accept(self)
        pair = self.accept(
        if pair is not None
            sock, addr = pai
            self.counter += 1
            name = "client" +      f.cou
            print('Incoming co      from         }'.
                  format(repr       name)
            self.broadcast(name        nnected")
            handler = ChatHandle (sock, name, self)
            self.clients[name] = handler

server = ChatServer('localhost', 7777)
asyncore.loop()
```

Tcl

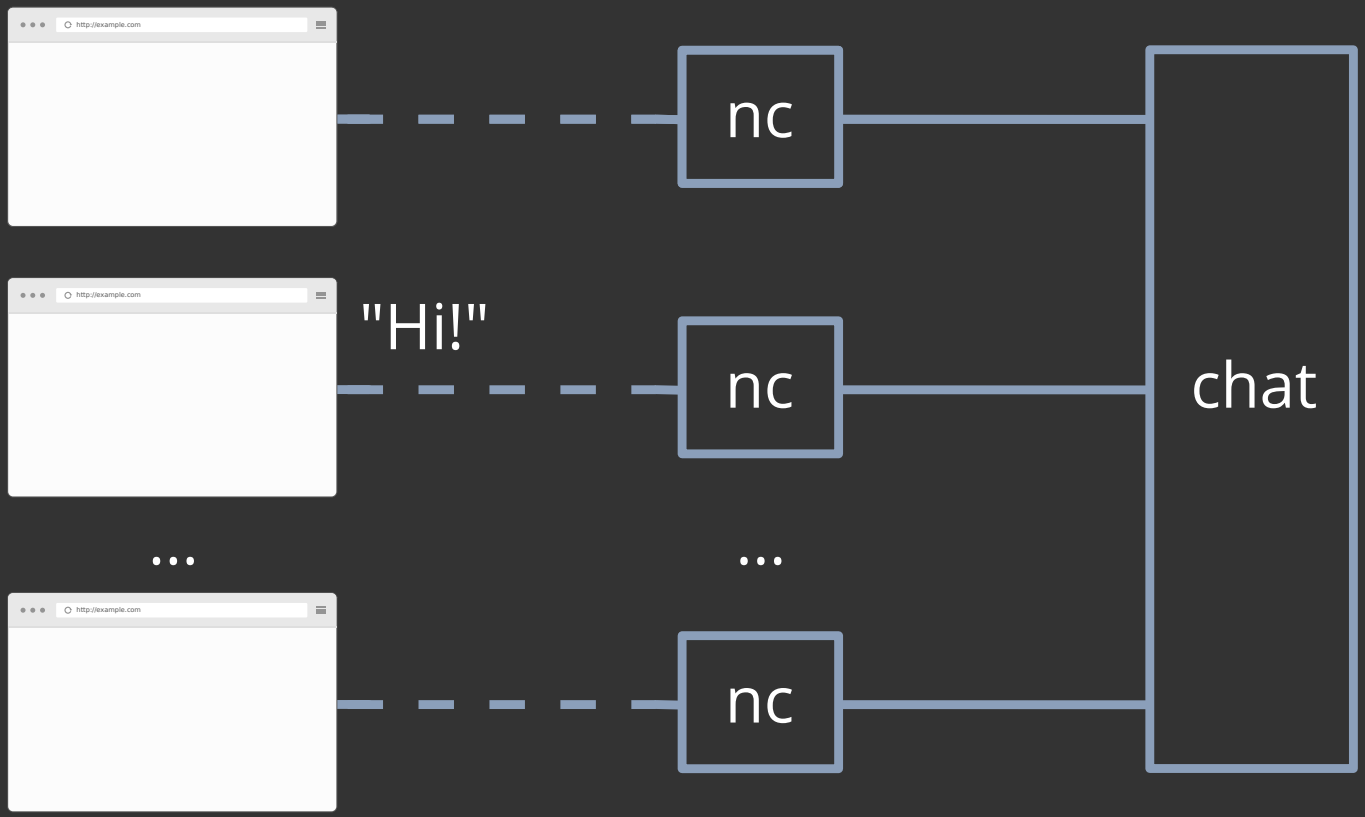
```
#!/usr/bin/env tclsh
namespace eval ::chat {
    variable count 0
    variable clients {}
}

proc ::chat::main
    socket -ser                    onne      ndler -myaddr 127.0.0.1 7777
           : for
}

pr      hat:            ion-hand  r {ch       lientAddr clientPort} {
        ::ch              nt
        name c
        pend ::c                  nnel $name

        s "Incoming connection from ('$clientAddr', $clientPort): $name"
        hat::broadcast "$name connected"

    fconfigure $channel -buffering line
    fileevent $channel readable [list ::chat::readable-handler $channel]
}

proc ::chat::broadcast message {
    puts $message
    foreach {otherChannel _} $::chat::clien
        puts $otherChannel $messag
    }
}

proc ::                  r ch
    set         e [         $::         ients
    if {        hanne        >= 0
                broadc        nder
    }
    if {[e      nnel]}
        ::        roadcas        derName disconnected"
        clo        nnel
        dic         ::chat::clients $channel

::chat::main
```

Never mind

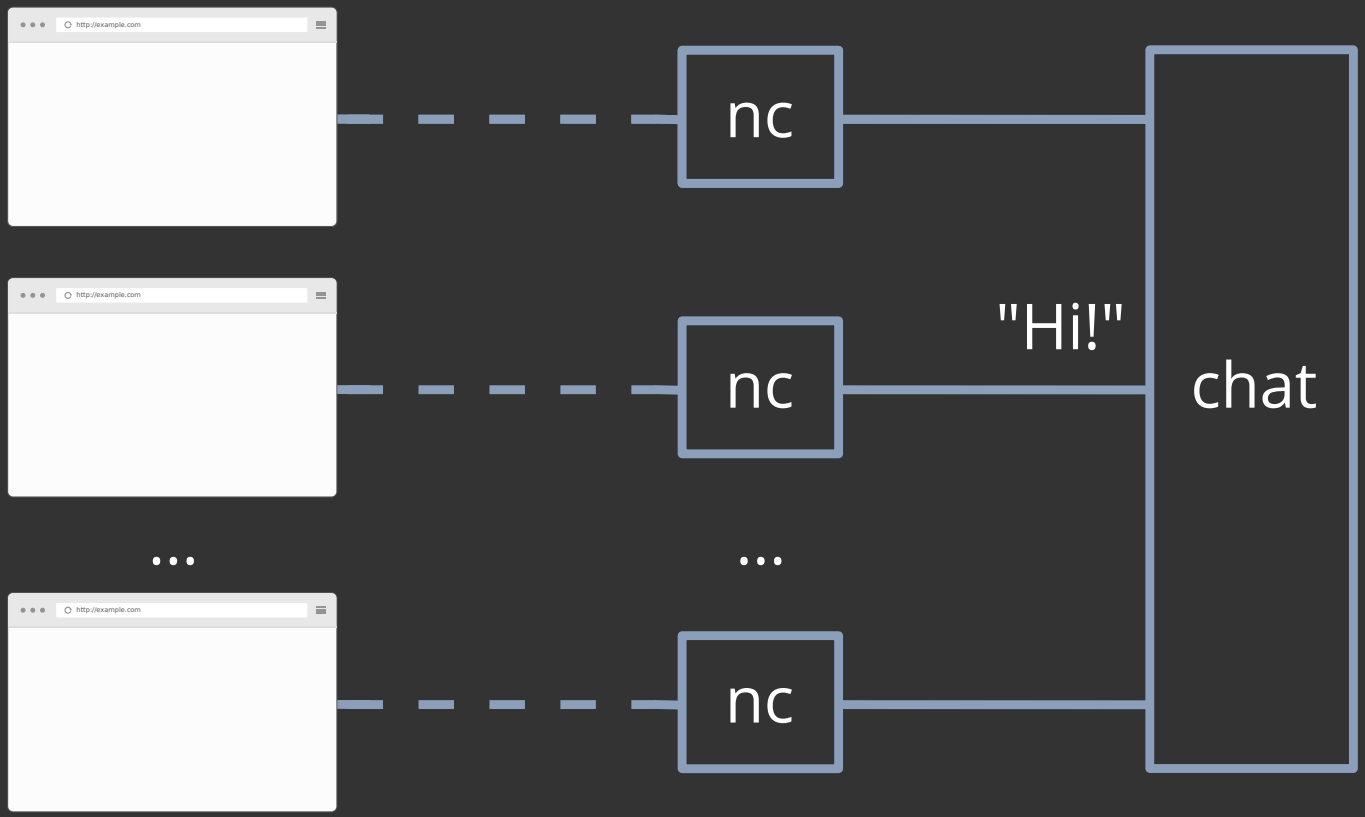
Example 3 — Server (3)

- The idea
  - Listen for connections on a TCP socket on localhost
  - Forward every message you get to all other clients
    - Prefix the message with the sender’s name

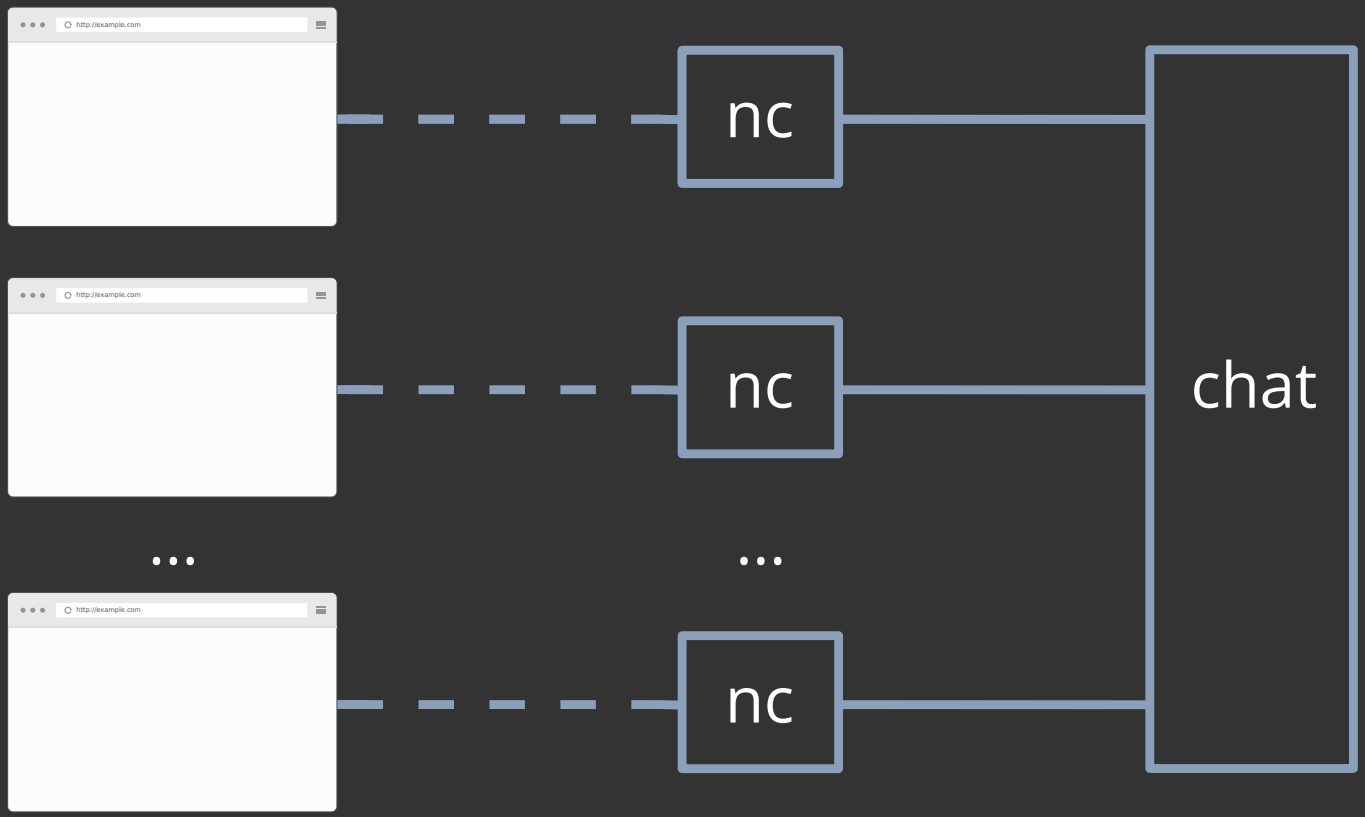
Example 3 — Data flow



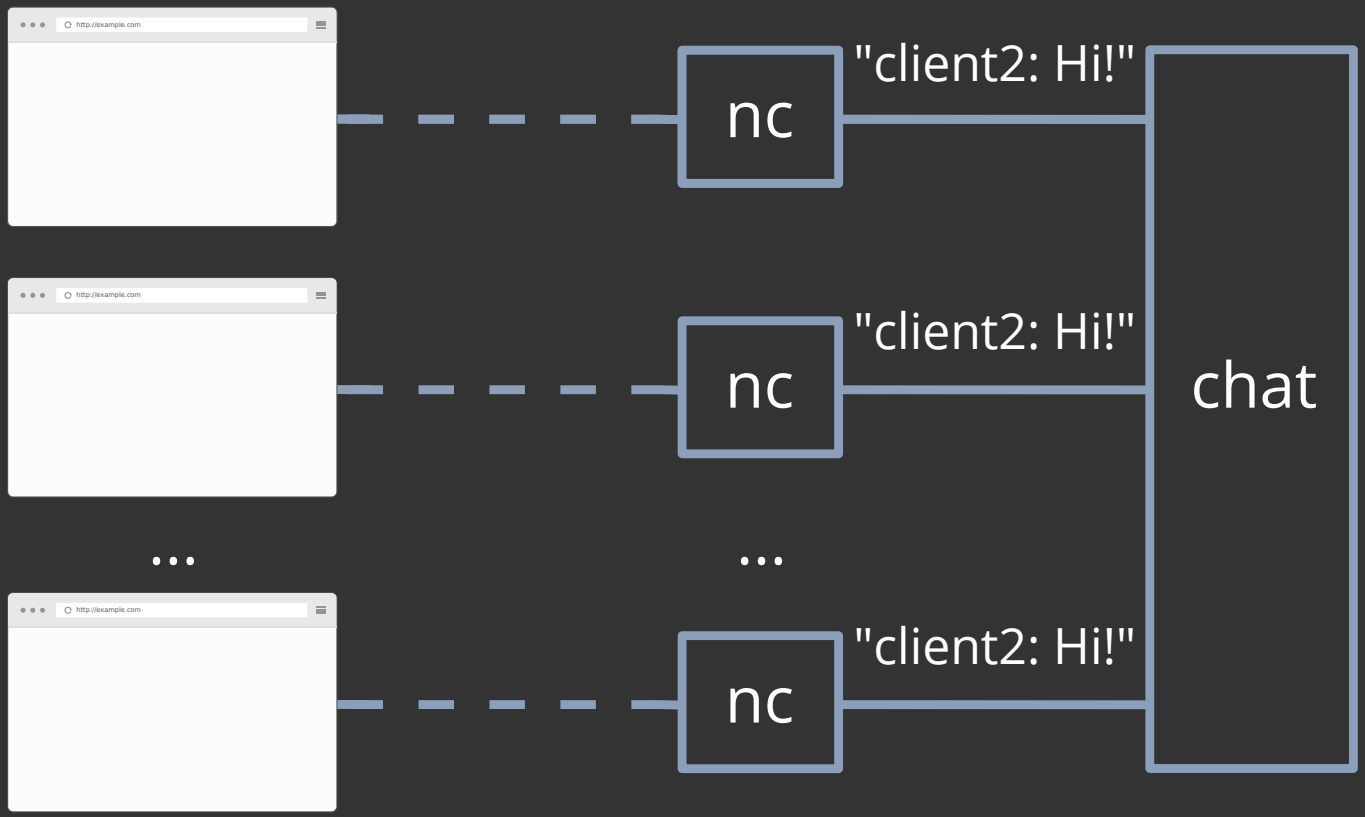
Example 3 — Data flow



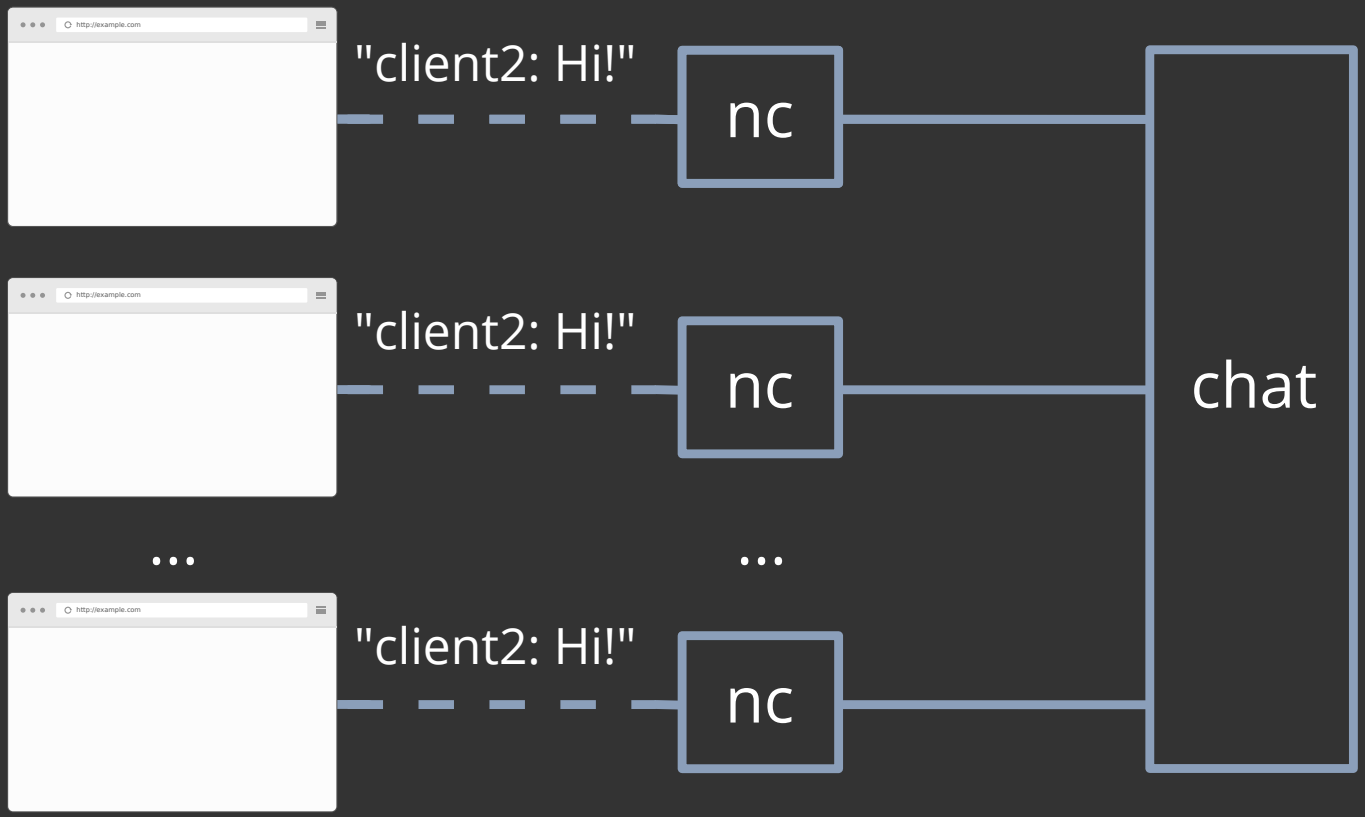
Example 3 — Data flow



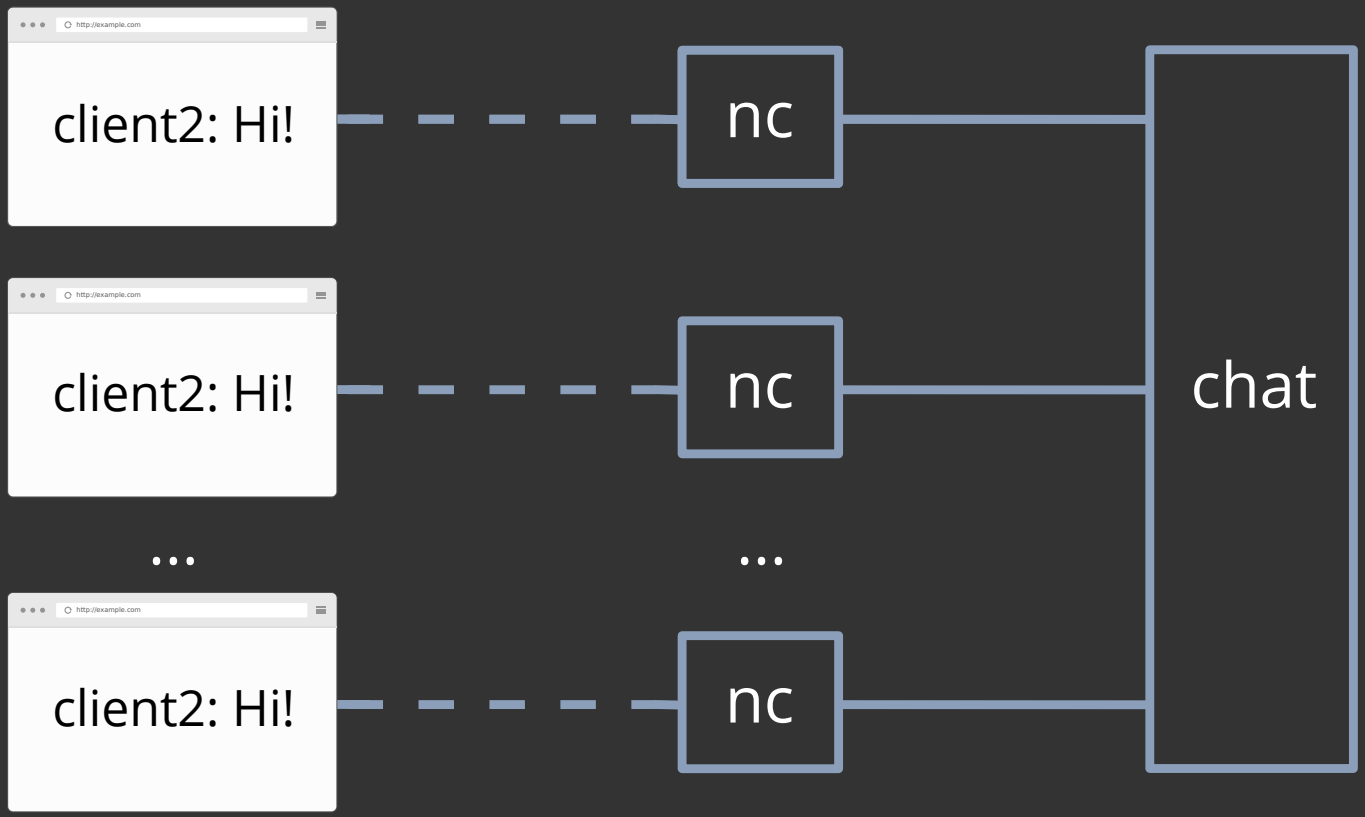
Example 3 — Data flow



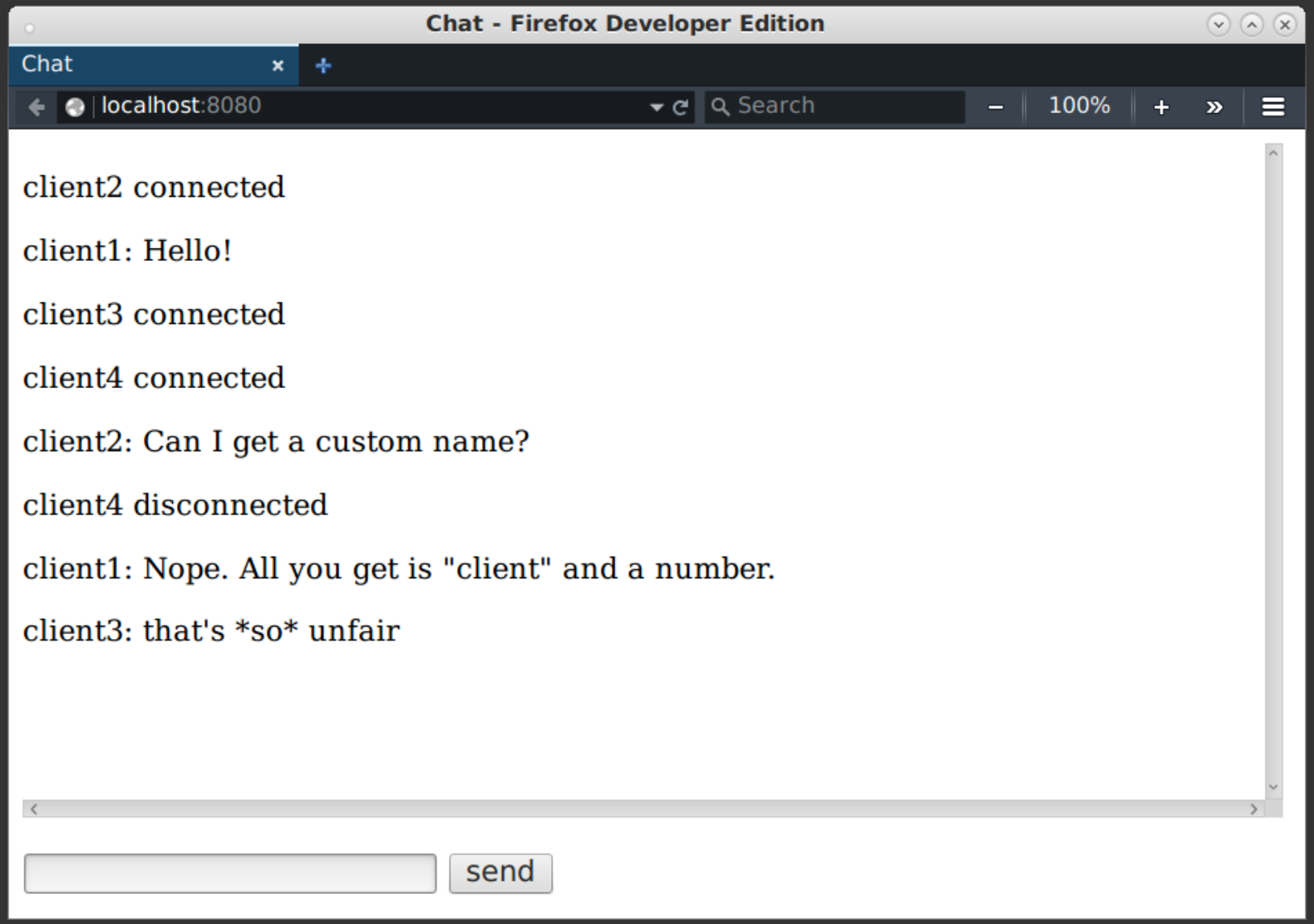
Example 3 — Data flow



Example 3 — Data flow



Example 3 — Screenshot



The Project

- A multiplayer game state replicator (aka “50% of an Agar.io server”)
- Tcl [4] on the server
  - Under 250 lines of code
- TypeScript [5] and Phaser [6] on the client
  - Type definitions for WebSocket objects from DefinitelyTyped

Observations on protocols

- No need to speak JSON (or worse yet, XML) both ways
- What worked
  - A simple plain text client-to-server protocol
    - One command per line
  - A JSON server-to-client protocol
    - Only send state changes to the client

The Recipe

websocketd

\+ netcat

\+ Tcl

(or your favorite scripting language)

---

easy rapid prototyping

Thank you!

References

1. http://websocketd.com/
2. See comment for an example. This matches the speaker's experience.
3. Look here or here.
4. https://en.wikipedia.org/wiki/Tcl
5. http://www.typescriptlang.org/
6. http://phaser.io/
7. Go to the GitHub repository for the talk to download the example code.