IIT Madras
BSc Degree

REST and APIs

API Design

- Web architecture - REST
- API Examples
- OpenAPI specification

Distributed Software Architecture

- Servers - Clients
- Standard “protocols” needed for communication
- Assumptions?
  - Server always on?
  - Server knows what client is doing?
  - Client authentication?
  - Network latency?

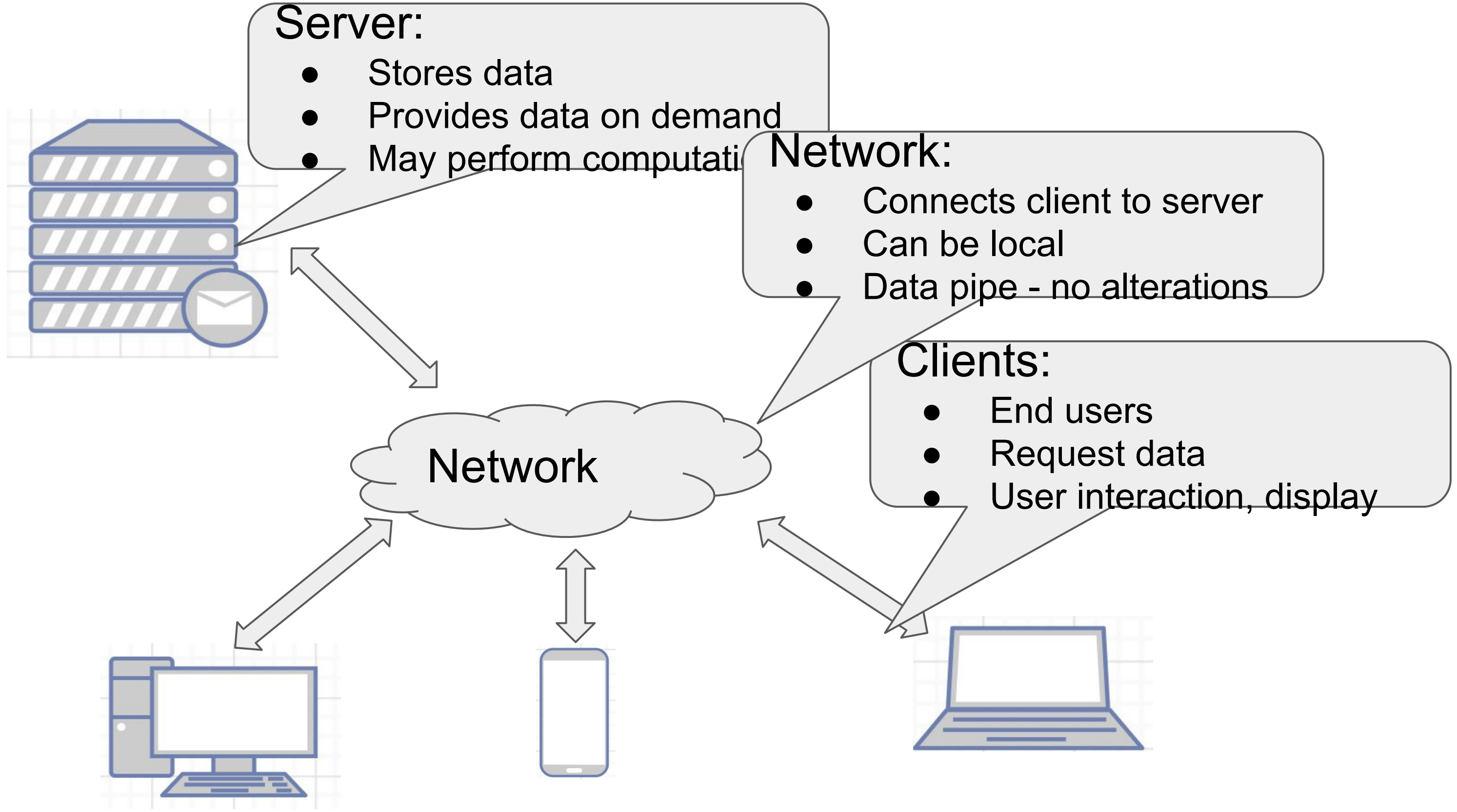
The Web

- Client - Server may be far apart
- Different networks, latencies, quality
- Authentication? Not core part of protocol
- State?
    - Server does not know what state client is in
    - Client cannot be sure what state server is in

Architecture for the Web

- Roy Fielding, PhD thesis 2000 UC Irvine
- “REpresentational State Transfer” - REST
    - Take into account limitations of the Web
    - Provide guidelines or constraints
- Software Architecture Style
    - Not a set of rules!

Constraint 1: Client - Server



Server:

- Stores data
- Provides data on demand
- May perform computati

Network:

- Connects client to server
- Can be local
- Data pipe - no alterations

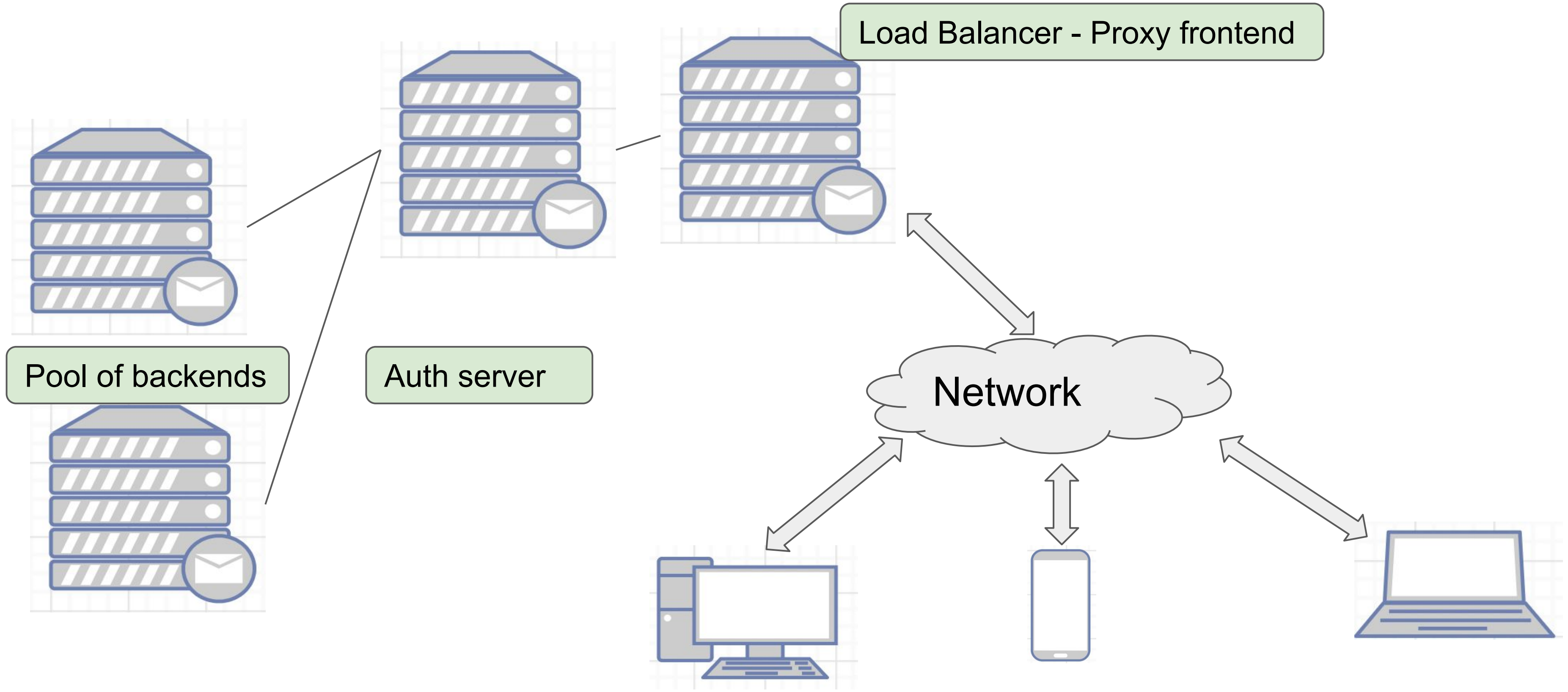
Clients:

- End users
- Request data
- User interaction, display

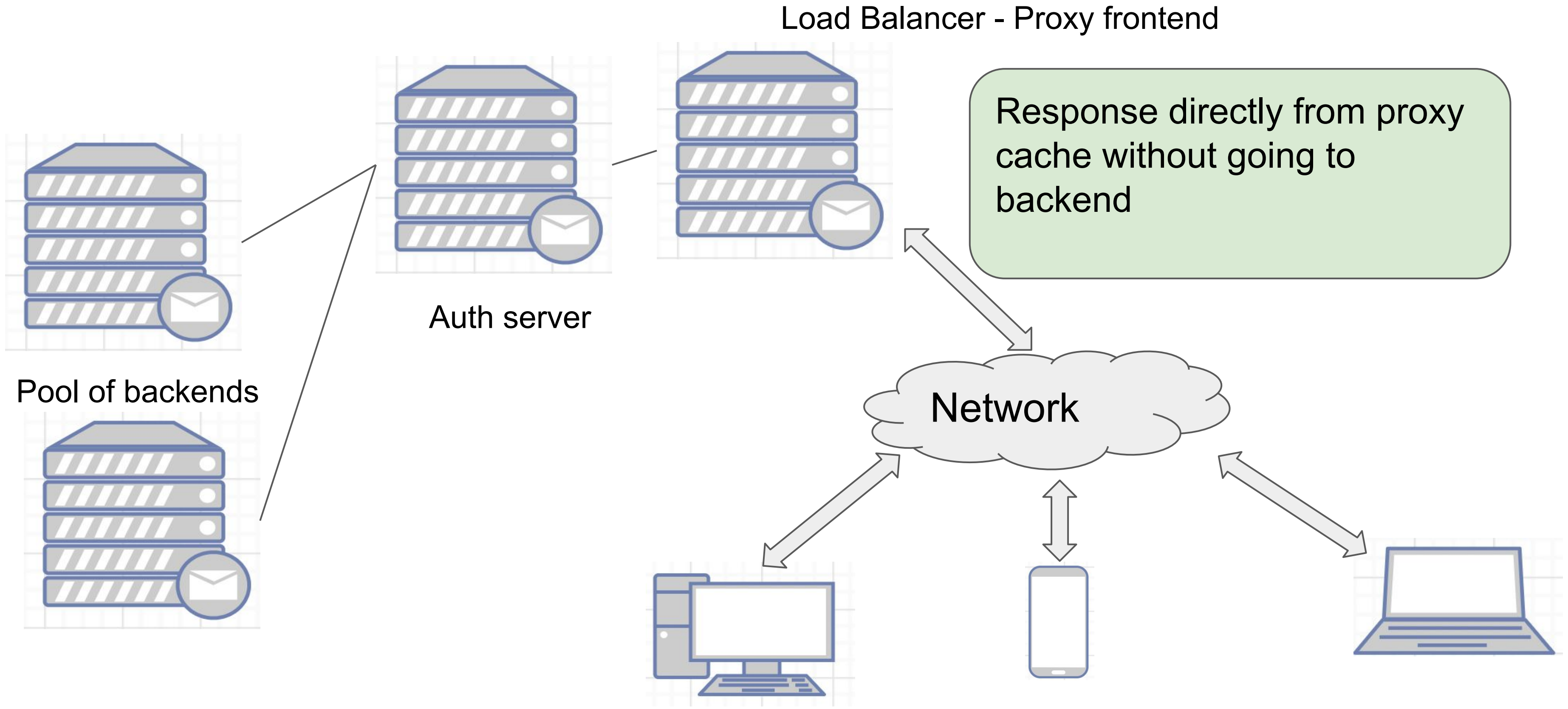
Constraint 2: Stateless

- Server cannot assume state of client:
  - which page are you looking at
  - is a request coming from an already logged in user just because of the address?
- Client cannot assume state of server:
  - did server reboot since last request?
  - is this request being answered by same server?

Constraint 3: Layered System



Constraint 4: Cacheability



Response directly from proxy cache without going to backend

Constraint 5: Uniform Interface

- Client and Server interact in a uniform and predictable manner
- Server exposes “resources”

Hypertext/media used to convey the available resources and functionality - can be discovered by client through hypertext information from server

(Optional) Constraint 6: Code on Demand

- Server can extend client functionality
    - Javascript
    - Java applets

Part of the overall architecture - these are not hard rules

REST

- REpresentational State Transfer

REST

- REpresentational State Transfer

What does that mean?

- State information between client and server explicitly transferred with every communication

Sequence

- Client accesses a Resource Identifier from server
    - Usually URI - superset of URL
    - Typically start from home page of application
    - No initial state assumed

Sequence

- Client accesses a Resource Identifier from server
  - Usually URI - superset of URL
  - Typically start from home page of application
  - No initial state assumed
- Resource Operation specified as part of access
  - If HTTP, then GET, POST etc.
  - Not fundamentally tied to protocol

Sequence

- Client accesses a Resource Identifier from server
  - Usually URI - superset of URL
  - Typically start from home page of application
  - No initial state assumed
- Resource Operation specified as part of access
  - If HTTP, then GET, POST etc.
  - Not fundamentally tied to protocol
- Server responds with new Resource Identifier
  - New state of system; new links to follow etc.

Sequence

- Client accesses a Resource Identifier from server
  - Usually URI - superset of URL
  - Typically start from home page of application
  - No initial state assumed
- Resource Operation specified as part of access
  - If HTTP, then GET, POST etc.
  - Not fundamentally tied to protocol
- Server responds with new Resource Identifier
  - New state of system; new links to follow etc.

State of interaction transferred back and forth

HTTP

- One possible protocol to carry REST messages
- Use the HTTP verbs to indicate actions
- Standardize some types of functionality

HTTP

- GET: Retrieve representation of target resource’s state
- POST: Enclose data in request: target resource “processes” it
- PUT: Create a target resource with data enclosed
- DELETE: Delete the target resource

Idempotent operations

- Repeated application of the operation is not a problem

Idempotent operations

- Repeated application of the operation is not a problem
- Example: GET is always safe - read-only operation

Idempotent operations

- Repeated application of the operation is not a problem
- Example: GET is always safe - read-only operation
- Example:
  - PUT: will always create the same new resource. If already exists, may give error

Idempotent operations

- Repeated application of the operation is not a problem
- Example: GET is always safe - read-only operation
- Example:
  - PUT: will always create the same new resource. If already exists, may give error
  - DELETE: can delete only once. may error on repeated deletion, but won’t change data

Idempotent operations

- Repeated application of the operation is not a problem
- Example: GET is always safe - read-only operation
- Example:
  - PUT: will always create the same new resource. If already exists, may give error
  - DELETE: can delete only once. may error on repeated deletion, but won't change data
- POST: May NOT be idempotent
  - Example: Add comment to blog - repeat will cause multiple copies

CRUD

- CRUD - database operations
- Typically a common set of operations needed in most web applications
    - Good candidate for REST based functionality

REST != CRUD

But they do work well together

Data Encoding

- Basic HTML: for simple responses
- XML: Structured data response
- JSON: simpler form of structured data

Data serialization for transferring complex data types over text based format

JSON

- JavaScript Object Notation
- Nested arrays:
    - Serialize complex data structures like dictionaries, arrays etc.

JSON

- JavaScript Object Notation
- Nested arrays:
    - Serialize complex data structures like dictionaries, arrays etc.

```json
{
  "firstName": "John",
  "lastName": "Smith",
  "age": 27,
  "address": {
    "streetAddress": "21 2nd Street",
    "postalCode": "10021-3100"
  },
  "phoneNumbers": [
    {
      "type": "home",
      "number": "212 555-1234"
    },
    {
      "type": "office",
      "number": "646 555-4567"
    }
  ],
  "children": [],
  "spouse": null
}
```

API data transfer format

- Input to API: text - HTTP
- Output: complex data types - JSON, XML, YAML etc.
  - JSON most commonly used
- Different from internal server representation
- Different from final view presentation

YAML

- Yet Another Markup Language - common alternative, especially for documentation and configuration