## Introduction

- Remote Procedure Call (RPC) is a high-level model for client-server communication.
- It provides the programmers with a familiar mechanism for building distributed systems.
- Examples: File service, Authentication service.

## RPC Model

<table>
<thead>
<tr>
<th>Blocking state</th>
<th>Executing state</th>
</tr>
</thead>
<tbody>
<tr>
<td>Call procedure and wait for reply</td>
<td>Receive request and start process execution</td>
</tr>
<tr>
<td>Resume execution</td>
<td>Send reply and wait for next execution</td>
</tr>
</tbody>
</table>

## Characteristics

- The called procedure is in another process which may reside in another machine.
- The processes do not share address space.
  - Passing of parameters by reference and passing pointer values are not allowed.
  - Parameters are passed by values.
- The called remote procedure executes within the environment of the server process.
  - The called procedure does not have access to the calling procedure's environment.
- No message passing or I/O at all is visible to the programmer.

## Features

- Simple call syntax
- Familiar semantics
- Well defined interface
- Ease of use
- Efficient
- Can communicate between processes on the same machine or different machines

## Limitations

- Parameters passed by values only and pointer values are not allowed.
- Speed: remote procedure calling (and return) time (i.e., overheads) can be significantly (1 - 3 orders of magnitude) slower than that for local procedure.
- Failure: RPC is more vulnerable to failure (since it involves communication system, another machine and another process).
  - The programmer should be aware of the call semantics, i.e., programs that make use of RPC must have the capability of handling errors that cannot occur in local procedure calls.
**Design Issues**

- Exception handling
  - Necessary because of possibility of network and nodes failures;
  - RPC uses return value to indicate errors;

- Transparency
  - Syntactic → achievable, exactly the same syntax as a local procedure call;
  - Semantic → impossible because of RPC limitation: failure (similar but not exactly the same);

**RPC Components**

- **Client**
  - **Client stub**: has same function signature as callee();
  - Allows same caller() code to be used for LPC and RPC

- **Communication Module**: Forwards requests and replies to appropriate hosts

- **Server**
  - **Dispatcher**: Selects which server stub to forward request to
  - **Server stub**: calls callee(), allows it to return a value

**Generating Code**

- Programmer only writes code for caller function and callee function
- Code for remaining components all generated automatically from function signatures (or object interfaces in Object-based languages)
  - E.g., Sun RPC system: Sun XDR interface representation fed into rpcgen compiler
- These components together part of a Middleware system
  - E.g., CORBA (Common Object Request Brokerage Architecture)
  - E.g., Sun RPC
  - E.g., Java RMI

**Marshalling**

- Different architectures use different ways of representing data
- Caller (and callee) process uses its own platform-dependent way of storing data
- Middleware has a common data representation (CDR) which is platform-independent
- Caller process converts arguments into CDR format
  - Called “Marshalling”
- Callee process extracts arguments from message into its own platform-dependent format
  - Called “Unmarshalling”
- Return values are marshalled on callee process and unmarshalled at caller process

**JSON-RPC**

- Remote procedure call protocol encoded in JSON.
  - It is a very simple protocol (and very similar to XML-RPC), defining only a handful of data types and commands.
  - Allows for notifications (data sent to the server that does not require a response)
  - Multiple calls to be sent to the server which may be answered out of order.
  - Invoked by sending a request to a remote service using HTTP or a TCP/IP socket (starting with version 2.0).
Example: adding

```javascript
var rpc = require('json-rpc2');
var server = rpc.Server.$create();

function add(args, opt, callback) {
    callback(null, args[0] + args[1]);
}

server.expose('add', add);
server.listen(8000, 'localhost');
```

```javascript
var rpc = require('json-rpc2');
var client = rpc.Client.$create(8000, 'localhost');

// Call add function on the server
client.call('add', [1, 2], function(err, result) {
    console.log('1 + 2 = ' + result);
});
```

Reference: https://github.com/pocesar/node-jsonrpc2
https://github.com/justmoon/node-jsonrpc2

### REST and HTTP
- The motivation for REST was to capture the characteristics of the Web which made the Web successful.
  - URI Addressable resources
  - HTTP Protocol
  - Make a Request – Receive Response – Display Response

- Exploits the use of the HTTP protocol beyond HTTP POST and HTTP GET
  - HTTP PUT, HTTP DELETE

### Resources
- The key abstraction of information in REST is a resource.
  - A resource is a conceptual mapping to a set of entities
    - Any information that can be named can be a resource
      - a document or image
      - a temporal service (e.g. "today's weather in Los Angeles")
      - a collection of other resources
      - a non-virtual object (e.g. a person)
  - Represented with a global identifier (URI in HTTP)
    - http://www.boeing.com/aircraft/747

### Verbs
- Represent the actions to be performed on resources
  - HTTP GET
  - HTTP POST
  - HTTP PUT
  - HTTP DELETE

### HTTP GET
- How clients ask for the information they seek.
- Issuing a GET request transfers the data from the server to the client in some representation
  - GET http://localhost/books
    - Retrieve all books
    - Retrieve book identified with ISBN-0011021
  - GET http://localhost/books/ISBN-0011021/authors
    - Retrieve authors for book identified with ISBN-0011021
HTTP PUT, POST, DELETE

- **POST** [http://localhost/books/](http://localhost/books/)
  - Content: {title, authors[], ...}
  - Creates a new book with given properties

- **PUT** [http://localhost/books/isbn-111](http://localhost/books/isbn-111)
  - Content: {isbn, title, authors[], ...}
  - Updates book identified by isbn-111 with submitted properties

  - Delete book identified by ISBN-0011

Representations

- How data is represented or returned to the client for presentation.

- Two main formats:
  - JavaScript Object Notation (JSON)
  - XML

- It is common to have multiple representations of the same data

Why is it called "Representational State Transfer"?

The client references a Web resource using a URL. A representation of the resource is returned (in this case as an HTML document). The representation (e.g., Boeing747.html) places the client application in a state. The result of the client traversing a hyperlink in Boeing747.html is another resource accessed. The new representation places the client application into yet another state. Thus, the client application changes (transfers) state with each resource representation → Representation State Transfer!

Architecture Style

<table>
<thead>
<tr>
<th>Request</th>
<th>Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTP GET</td>
<td>URL 1</td>
</tr>
<tr>
<td>HTTP POST</td>
<td>URL 1</td>
</tr>
<tr>
<td>HTTP DELETE</td>
<td>URL 1</td>
</tr>
</tbody>
</table>

Example: REST for bears

<table>
<thead>
<tr>
<th>Route</th>
<th>HTTP Verb</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>/api/bears</td>
<td>GET</td>
<td>Get all the bears.</td>
</tr>
<tr>
<td>/api/bears</td>
<td>POST</td>
<td>Create a bear.</td>
</tr>
<tr>
<td>/api/bears/bear_id</td>
<td>GET</td>
<td>Get a single bear.</td>
</tr>
<tr>
<td>/api/bears/bear_id</td>
<td>PUT</td>
<td>Update a bear with new info.</td>
</tr>
<tr>
<td>/api/bears/bear_id</td>
<td>DELETE</td>
<td>Delete a bear.</td>
</tr>
</tbody>
</table>

Example: Create a bear

```javascript
var express = require('express');
var app = express();
var router = express.Router();
var bodyParser = require('body-parser');
var bears = [];

router.route('/bears').
  .post(function(req, res) {
    var bear = {};
    bear.name = req.body.name;
    bears.push(bear);
    res.json({ message: 'Bear created!' });
  });

// all of our routes will be prefixed with /api
app.use('/api', bodyParser.json(), router);
app.listen(8000);
```
Try REST API

- Chrome plugins with REST clients functionality are available, e.g., Postman, DHC

Get all bears (1)

```javascript
import React, { Component } from 'react';
import axios from 'axios';
import _ from 'lodash';

const URL = 'http://localhost/api/bears';

class Bear extends Component {
  constructor(props) {
    super(props);
    this.state = { data: {} }
  }

  componentDidMount() {
    axios.get(URL)
      .then(response => {
        this.setState({data: response.data});
        console.log(response.data)
      })
      .catch(error => {
        console.log(error)
      });
  }

  renderBears() {
    return _.map(this.state.data, bear => {
      return (<li className="list-group-item" key={bear.id}>
        {bear.id + 1}. {bear.name}, {bear.weight}
      </li>);
    });
  }

  render() {
    return (<div>
      <h2>Bear Profile</h2>
      <ul className="list-group">
        {this.renderBears()}
      </ul>
    </div>);
  }
}
```

Cross-Origin Resource Sharing (CORS)

- Allow APIs to be called from different domains
- npm install cors

Get all bears (2)

```javascript
renderBears() {
  return map(this.state.data, bear => {
    return (<li className="list-group-item" key={bear.id}>
      {bear.id + 1}. {bear.name}, {bear.weight}
    </li>);
  });
}
```

Axios: Post and Delete example

```javascript
axios.post('http://localhost/api/bears', {
  name: 'Fred',
  weight: 123
}).then((response) => {
  console.log('Create a bear: ' + response);
}).catch((error) => {
  console.log(error);
});

axios.delete('http://localhost/api/bears/5').then((response) => {
  console.log('Delete:' + response);
});
```
Redux

- Redux is a predictable state container for JavaScript apps.
- Write applications that behave consistently, run in different environments (client, server, and native), and are easy to test.

- Redux divides a component into several types:
  - Components (View)
  - Actions (Event)
  - Reducers (Data)


Component (components/bear_index.js)

```jsx
class BearIndex extends Component {
  constructor(props) {
    super(props);
  }

  componentDidMount() {
    this.props.fetchBears();
  }

  renderBears() {
    return _.map(this.props.bears, bear => {
      return (<li className="list-group-item" key={bear.id}>
        {bear.id+1}. {bear.name}, {bear.weight}
      </li>);
    });
  }

  render() {
    return (<div>
      <h2> Bear Profile </h2>
      <ul className="list-group">
        {this.renderBears()} 
      </ul>
    </div>);
  }
}

function mapStateToProps(state) {
  return { bears: state.bears };
}

export default connect(mapStateToProps, {fetchBears})(BearIndex);
```

Actions (actions/index.js)

```jsx
import axios from 'axios';

export const FETCH_BEARS = 'fetch_bears';
const ROOT_URL = 'http://localhost/api/bears';

export function fetchBears() {
  const request = axios.get(ROOT_URL);
  return {
    type: FETCH_BEARS, 
    payload: request
  };
}
```

Reducers (reducers/index.js)

```jsx
import { combineReducers } from 'redux';
import { reducer as formReducer } from 'redux-form';
import BearsReducer from './reducer_bears';

const rootReducer = combineReducers({
  bears: BearsReducer
});

export default rootReducer;
```
reducers/bear_reducer.js

```javascript
import _ from 'lodash';
import {FETCH_BEARS } from '../actions';

export default function (state = {}, action) {
  switch (action.type) {
    case FETCH_BEARS:
      return _.mapKeys(action.payload.data,'id');
    default:
      return state;
  }
}
```

Main page - Router

```javascript
import React, { Component } from 'react';
import { Provider } from 'react-redux';
import { createStore, applyMiddleware } from 'redux';
import { BrowserRouter, Route, Switch } from 'react-router-dom';
import reducers from './reducers';
import BearIndex from './components/bear_index';
const createStoreWithMiddleware = applyMiddleware(promise)(createStore);
```

References

- "Advanced Operating Systems: RPC", Ajay Kotangur
- "RPCs and Concurrency Control", Indranil Gupta, 2014
- "Node:jsonrpc2", https://github.com/pocesar/node-jsonrpc2
- "Representational State Transfer (REST): Representing Information in Web 2.0 Applications", Emilio F Zegarra
- "Build a RESTful API Using Node and Express 4", https://scotch.io/tutorials/build-a-restful-api-using-node-and-express-4