JSMPI: Building Parallel Applications in the Browser Environment

Thanasis Petsas, Elias Athanasopoulos and Sotiris Ioannidis
Institute of Computer Science
Foundation for Research and Technology, Hellas
{*petsas, elathan, sotiris*}@*ics.forth.gr*

Abstract

We take advantage of the massive adoption of the web browser as the application for carrying out a plethora user activities. Considering there are millions of web users on-line at any given moment, a collection of web browsers seems ideal for running distributed and parallel computations. In this paper, we design, implement and evaluate JSMPI; a framework for building traditional Message Passing Interface (MPI) applications that can run in the web browser environment. JSMPI was designed so as to maintain the semantics of the traditional MPI. By using JSMPI, an MPI program, with little code modifications, can be executed by a collection of web browsers, as it would on a native MPI environment.

1 Introduction

The web browser is capable of running multiple, feature-rich applications. The web, composed of a series of advanced and mature technologies, such as HTTP, JavaScript and XML, is a modern platform for building applications, that were traditionally perceived as the *de facto* applications that complement a modern operating environment (Gmail, Google Docs, Picasa) Major IT vendors have built complete operating systems and platforms based solely on the use the web (ChromeOS, CR-48).

According to work done in [5], the estimated mean time of someone surfing the web is 74 minutes. This estimation was conducted 5 years ago, and we expect an increased mean surf time today. Thus, the web browser seems the ideal platform for utilizing for distributed computation. Commodity devices have also been proposed for outsourcing computation in the past [2, 1, 7]. However, this is first time a traditional framework for building parallel applications, such as MPI is built in the web browser.

We make the following contributions:

- We design, implement and evaluate JSMPI, an MPI version for the web browser. Our evaluation shows that traditional computational problems, such as the calculation of Pi, demonstrate similar speedup in JSMPI as with traditional MPI.
- We focus on the semantics of JSMPI and provide an identical API as in MPI. An MPI program written in C, with minimal modifications, can be easily transformed to a JSMPI program written in JavaScript, and run *as is* on a collection of browsers.

2 Design and Impementation

JSMPI consists of two main components: the *JSMPI Client* that is a browser extension containing all the necessary code for writing parallel applications, and the *JSMPI Cache* which is a directory server similar to the one used by Tor [4].

The **JSMPI Client** is the core component of our architecture as it contains all the functions that are needed for building parallel applications in the browser environment. Our intention was to make the Client easily deployable and readily available to as many Internet users as possible. For these reasons we chose the web browser as a platform to implement our framework. We have designed our framework as a Firefox extension. Firefox is a widely-used Internet browser available in many different platforms with strong support in third-party plugins. Users that want to participate in the JSMPI network simply install the JSMPI extension via one-click using the build in installer of Firefox. Our Client is written entirely in JavaScript using only the Mozilla API. This ensures that Firefox is the only requirement for someone wanting to use our framework. Furthermore, for systems with low resources it is possible to use XULrunner instead of Firefox, a runtime environment developed by the Mozilla Foundation to provide a common back-end for XUL applications.

Function	Request	Headers
$Ping$	GET	X-PORT
$List$	GET	-
$GetMyIP$	GET	-

Table 1: Set of JSMPI node functions used for interaction with the Cache

JSMPI Clients are separated in two categories: the *slaves* and the *masters*. Both of have to start by registering with the JSMPI Cache (as described later). Afterwards a slaves wait to receive a URL containing the location of the application's source code from a master. A master is responsible to send such a URL to slaves (obtained from the JSMPI Cache). Then, the slaves send an acknowledgment to master informing that they have received the URL and start running the application. When the master receives the acknowledgments from all the slaves it also starts executing the application. This process is hidden from the volunteer users that participate in the network (slave nodes), as well as from developers that have written and test their applications (master nodes), since they are implemented as inner functions of the framework. All the communication between the JSMPI nodes, as well as between nodes and Caches is done via HTTP requests. HTTP requests are sent by using the *XMLHttpRequest* object, which is prototyped in all browsers. For handling HTTP requests from other nodes or the Cache, each node implements an HTTP server. Currently the server used by our framework is a modified version of the server used for testing plugins in the Mozilla Development Center. Also, we use the native support for threads to parallelize tasks that require network communication and thus impose delay penalties. To overcome possible problems in case nodes run behind a NAT or inside a VPN network, where it would be difficult to discover its real IP address, nodes ask for their IP addresses from the Cache at startup.

The second component of our system is the **JSMPI Cache**. JSMPI Caches are similar to the directory servers that Tor uses to preserve a distributed information of known routers and their current state across the Tor network. They comprise the main registries of the JSMPI network and share a list with the currently active nodes. JSMPI Caches are simple HTTP servers implemented in Python based on the *BaseHTTPServer* module. They include a fairly small processing logic since their functionality is limited to registering JSMPI nodes in the network, populating the list of nodes and informing nodes for their IP addresses when they reside behind a NAT firewall or inside a VPN network.

All the communication between the nodes and the Caches happens through simple HTTP GET requests. In Table 1 we list all the functions (HTTP requests with the appropriate headers) that a node may use to communicate with a Cache. When a node wishes to participate in the network, it has to register itself with the Cache through a specific HTTP request called a ***Ping*** request. The node sends a *Ping* request to a Cache and specifies its port in a header (X-PORT header). Whenever the Cache receives such a request identifies the node as alive, saves its pair of IP address and port in a structure and replies with an acknowledgment. This response helps the node to understand if the Cache is also alive. Moreover, *Ping* requests can be exchanged between nodes for the same purpose: a node *A* can use a *Ping* request to identify if a node *B* is alive. Furthermore, both nodes and Caches make periodical *Ping* requests to ensure for the availability of the each other. Another communication function that a node may use is the ***List*** request. Using this request a node can be informed from the Cache about currently alive nodes that participate in the network. Also, *List* requests may be sent between nodes so that the network can keep operating in case that a set of Caches becomes unreachable. Finally, a node maybe want to discover of its real IP address when lying behind a NAT box or a VPN network. To achieve this, it can use the ***GetMyIp*** function sending a *GetMyIP* request to the Cache. The response will contain its public IP address.

2.1 API

We now describe the basic API of our framework. There is a core set of functions that all traditional implementations of MPI support. We start by presenting the specification (and behavior) of this set of functions in our framework and continue with a set of collective communication functions for more elaborated programs.

2.1.1 Basic MPI Functions

`JSMPIClient` This is the constructor of the JSMPIClient class. By calling this function the browser becomes a JSMPI slave and all the necessary procedures take place such as the initialization of the HTTP server to handle the requests, the notification of its presence to the Cache, the look up of the currently available nodes *etc.*.

`finalize` This function is the destructor of the JSMPILCient and terminates all the MPI processing. It must be called at the end of the MPI code. After the `finalize` function call, only non-MPI code is permitted. Any MPI calls made after a `finalize` function call will cause an error.

`getClientSize` Through this function a node can be informed about the currently available nodes participating in the network. The result comes from a *Ping* request made from the node to the Cache.

`getClientId` If a node needs to get its identifier it calls this function. The identifier of a node in JSMPI is the pair of the IP address and the port of the HTTP server that runs inside the node. A *GetMyIP* request is made to the Cache for the node to be discover its public IP address when is behind a NAT or inside a VPN network.

`send` This function is used for sending messages between nodes in our framework. The content of the message can be of any type supported by JavaScript. We apply a serialization process for every message content to be able to be transmitted via HTTP. The marshalling of a message content is done via *toSource()* JavaScript's native method.

`receive` This function is used to receive a message from a node. The message is being received through the HTTP server that every Client implements. When a Client receives a message has to deserialize its contents. The unmarshalling of a message content is achieved through *eval()* JavaScript's native method.

2.1.2 Collective communication functions

In all the functions below, a group of nodes may refers to all the nodes participating to a parallel application or a subset of them that is defined by an optional input argument.

`broadcast` broadcasts a message from one node to a group of nodes in a parallel application. After the end of this function each node of the group will have received the same data.

`gather` gathers together messages from a group of participating nodes into an array object at the Client where the call takes place.

`allGather` gathers data from a group of participating nodes and distributes them again to all these nodes. To collect the data from the nodes `allGather` uses the `gather` function. Similarly, to distribute them to all the nodes, it uses the `broadcast` function.

`scatter` scatters data from one node to a group of participants. The data is being cut into chunks and each chunk targets a particular node.

`allScatter` gathers data from a group of participants and scatters them back to all of them. To collect the data from the nodes `allScatter` uses the `gather` function. Similarly, to intersperse them to all the nodes, it uses the `broadcast` function.

`reduce` gathers data from a group of participant nodes and reduces them through a specific operation (add, mul, min, max *etc.*) to a single value.

`allReduce` reduces data from a group of participant nodes and distribute the result back to all of them using the `broadcast` function.

2.2 Examples

In 1 we show an example of a simple parallel application that computes the sum of an array of numbers written (a) for our framework, and (b) for traditional MPI.

In the **first part** of the example we call the constructor of JSMPIClient. The two arguments that taken as input are defined in a configuration file and can be changed on demand. The control flow will meet the **second part** of the program (if block) if the node is a slave. In this case the node is waiting to receive data from the master through a `receive` call. The master that is used as argument in the `receive` function is a global variable defined inside the framework. Its initial value has been set when a master node chose to reserve this Client, so as to participate in this parallel application. When the slave receive the message which has a chunk of the numbers array, will compute the sum of them and will proceed to a `send` function call sending the result back to the master. In the opposite case, if the node is a master, the control flow will reach the **third part** of the program (else if block). In this part the master assign to the variable numbers an array that parses from file named as "array.txt". Then a `scatter` call will follow to send the array to all the slaves. Finally the master will compute the sum for its own chunk and will gather and aggregate together all the slaves' sum results. Finally in the **fourth part**, both a slave and a master will call the `finalize` function to stop the parallel application.

3 Evaluation

To evaluate our framework we developed two parallel applications. The first is a simple parallel PI calculation and the second one is a distributed search of an element inside a *JavaScript* array. In order for our analysis to conform with the dynamic nature, latency and heterogeneity of the Internet, all the experiments were conducted on PlanetLab [3].

Figures 2 and 3 show the execution time of the pi calculation and distributed search application respectively for an increasing number of web browsers. We immediately notice that the total execution time scales as we add more nodes to the network. The results are better for the pi calculation than for distributed search application as the second involves more intensive communication operations with larger amount of data. Figure 4 illustrates the corresponding speedup ratios achieved in our experiments. One can see that while speedup is possible, due to communication limitations this is not linear. The maximum speed-up for the pi calculation reaches 6.4 and for the distributed search it reaching 2.6 times when the network consists from 32 browsers. The speedup ratios are less for the distributed search application than for the pi

```
// part 1
var node = new JSMPIClient(cache, port);
// part 2
if (mode == "slave") {
  var msg = node.receive(master);
  var sum=0;
  for (var i=0; i<msg.length; i++) {
    sum += msg[i];
  }
  node.send(master, sum);
}


// part 3
else if (mode == "master") {
  var numbers = readFile("array.txt");
  var master_chunk =
  node.scatter(numbers);
  var sum = 0;
  for (var i=0; i<master_chunk.length; i++) {
    sum += master_chunk[i];
  }
  var slaves_sums = node.gather();
  for (var i=0; i<slaves_sums.length; i++) {
    sum += slaves_sums[i];
  }
}

// part 4
node.finalize();
```

(a)

```
// part 1
MPI_Init(&argc, &argv);
// part 2
if (mode !=0) {
  MPI_Recv (&msg, 1, MPI_INT, 0,
      tag, MPI_COMM_WORLD, &status);
  int i, sum=0;
  for (i=0; i<array_length(msg); i++) {
    sum += msg[i];
  }
  MPI_Bsend (&sum, 1, MPI_INT, 0,
                  tag, MPI_COMM_WORLD);
}
// part 3
else { // master
  int *numbers = read_arrayfile("array.txt");
  int *master_chunk = split_array(&numbers);
  MPI_Scatter( numbers, CHUNK_SIZE, MPI_INT,
      rbuf, CHUNK_SIZE, MPI_INT, root, comm);
  int i, sum=0;
  for (i=0; i<array_sz(master_chunk); i++) {
    sum += master_chunk[i];
  }
  MPI_Gather(slaves_sums, MPI_INT, MPI_INT,
          rbuf, MPI_INT, MPI_INT, root, comm);
  for (i=0; i<array_sz(slaves_sums); i++) {
    sum += slaves_sums[i];
  }
// part 4
MPI_Finalize();
```

(b)

Figure 1: A piece of code that computes the sum of array numbers expressed in JSMPI framework (a) and in traditional MPI in the C language (b).

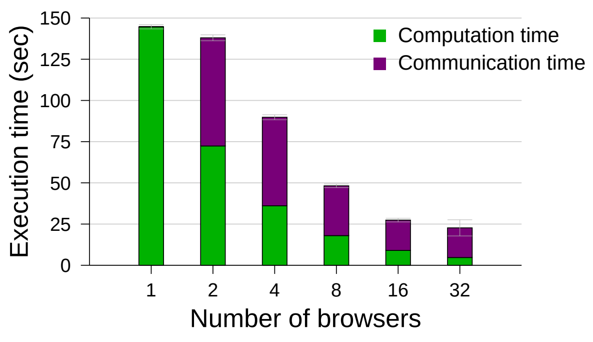

Figure 2: Execution time of pi calculation for an increasing number of browsers tested on PlanetLab

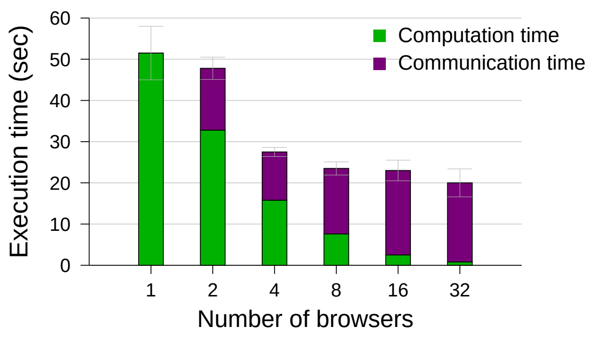

Figure 3: Execution time of distributed search for an increasing number of browsers tested on PlanetLab

calculation due to communication overhead. It is worth mentioning that tried to used the less loaded PlanetLad nodes in terms of CPU utilization while conducting our experiments.

We also measured the latency between two JSMPI nodes in PlanetLab. Figure 5 shows the results of our experiments for different message sizes between 1KB and 32MB on a double-logarithmic scale.

4 Related Work

Yue et al. [8] leverage the power of Ajax and the end-user extensibility of modern Web browsers to implement a simple and practical framework for Real-time Collaborative Browsing (RCB).

In [6], McKinley et al. develop an object-oriented middleware framework called *Pavilion* which allows a

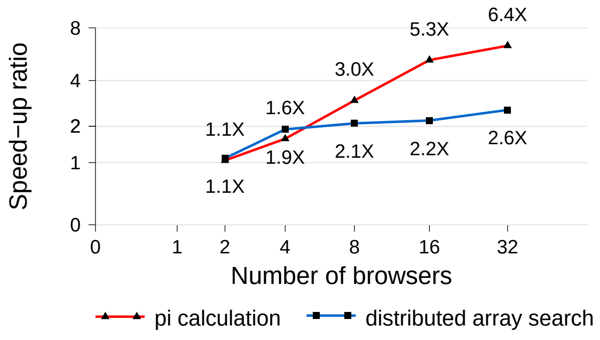

Figure 4: Speed-up ratios of the PI calculation and distributed array search for an increasing number of browsers tested on PlanetLab

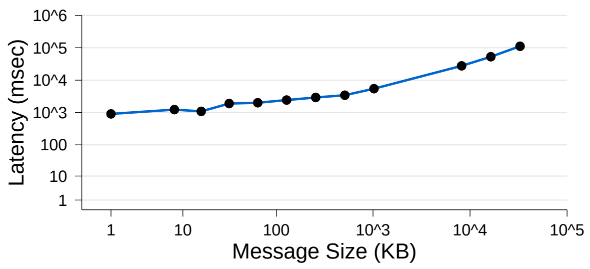

Figure 5: Latency of a one-to-one communication via JSMPI in PlanetLab.

developer to implement collaborative web-based applications using its current functionality or by extending it.

In [2], Chandra et al. propose a different approach to the current *"Pay-as-you-go"* cloud model of strong guarantees and highly centralized infrastructures. Their idea is to build clouds by using distributed voluntary resources donated by end-user hosts. They call their concept *"Nebulas"*, more decentralized and less-managed clouds. Our approach is more similar to the *Nebulas* model as our framework can be embedded in the web browser of any Internet user globally (high decentralization) and it is of no cost since it is provided-donated by end-users.

5 Conclusions

We presented the design and implementation of JSMPI, an MPI-compatible programming model, based on the JavaScript language and the browser execution platform. Moreover we have enhanced the Mozilla Firefox browser through an extension so as to support our framework. Our goal was to lay a foundation of a network comprised of web browsers living on Internet-users' machines that is capable of executing distributed and parallel applications.

To demonstrate the effectiveness of our framework we conducted our experiments on the PlanetlLab testbed that resembles the dynamic nature and heterogeneity of the Internet. Our experimental results show that the execution of the programs scales reasonably over a growing set of web browsers.

References

[1] David P. Anderson. Boinc: A system for public-resource computing and storage. In *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, GRID '04, pages 4–10, Washington, DC, USA, 2004. IEEE Computer Society.

[2] A. Chandra and J. Weissman. Nebulas: Using Distributed Voluntary Resources to Build Clouds. In *USENIX 2009 HotCloud Conference Proceedings*, June 2009.

[3] Brent Chun, David Culler, Timothy Roscoe, Andy Bavier, Larry Peterson, Mike Wawrzoniak, and Mic Bowman. PlanetLab: An Overlay Testbed for Broad-Coverage Services. *ACM SIGCOMM Computer Communication Review*, 33(3):00–00, July 2003.

[4] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. In *Proceedings of the 13th USENIX Security Symposium*, August 2004.

[5] V. T. Lam, S. Antonatos, P. Akritidis, and K. G. Anagnostakis. Puppetnets: misusing web browsers as a distributed attack infrastructure. In *Proceedings of the 13th ACM conference on Computer and communications security*, CCS '06, pages 221–234, New York, NY, USA, 2006. ACM.

[6] P. K. McKinley, A. M. Malenfant, and J. M. Arango. Pavilion: a middleware framework for collaborative web-based applications. In *GROUP '99: Proceedings of the international ACM SIGGROUP conference on Supporting group work*, pages 179–188, New York, NY, USA, 1999. ACM Press.

[7] Shouhuai Xu and Moti Yung. Socialclouds: Concept, security architecture and some mechanisms. In *INTRUST*, pages 104–128, 2009.

[8] Chuan Yue, Zi Chu, and Haining Wang. Rcb: a simple and practical framework for real-time collaborative browsing. In *USENIX'09: Proceedings of the 2009 conference on USENIX Annual technical conference*, pages 29–29, Berkeley, CA, USA, 2009. USENIX Association.