# Cross-context Web browser communication with unified communication models and context types

Ivan Zuzak, Marko Ivankovic, Ivan Budiselic
School of Electrical Engineering and Computing, University of Zagreb, Zagreb, Croatia
E-mail: {izuzak, ivankovic.42, ibudiselic}@gmail.com

*Abstract* - **Modern Web applications are developed and executed as a set of interacting browser contexts, such as windows, frames and background workers. Cross-context communication systems have been rapidly evolving to support this interaction, but are still behind modern Web application demands. In this paper we present Pmrpc, a cross-context communication system with several novel properties important for future Web applications. First, the system provides three communication models, message-based, remote procedure call and publish-subcribe, and exposes them through a single unified programming interface. Second, the system enables communication between both window-type contexts and worker-type contexts using the same unified programming interface. Third, the system enables dynamic discovery of contexts. We present the architecture of Pmrpc, based on secure message-oriented browser primitives introduced in the HTML5 group of standards. Lastly, we compare the execution times of achieving specific application goals when using Pmrpc to those of HTML5 native browser primitives. We show that although Pmrpc is slower than native primitives, the reduction in performance is not significant and the system is still usable in real-world Web applications.**

## I. INTRODUCTION

Web browsers are rapidly evolving in response to the increasing demands of modern Web applications. One trend is seen in developing Rich Internet Applications [1] which execute in a Web browser as a composition of contexts, such as browser frames. This trend is mainly visible in mashup Web applications [2], complex widget-based applications, such as Geppeto [3], and applications which use GUI-less threads for background processing [4] [5]. Therefore, Web browsers are becoming environments for execution of Web applications and support the interaction between execution contexts, similar to operating systems executing multi-process applications and supporting inter-process communication [6].

Although cross-context communication in Web browsers has historically been a difficult task due to strict browser security policies, many systems that enable such communication were developed. However, most of the developed systems are unresearched with regard to aspects unrelated to security and little research was dedicated to future Web applications requirements for cross-context communication. Consequently, Web researchers and developers face a complex ecosystem in which it is often difficult to comprehend the capabilities worth observing, discern each system's capabilities and evaluate benefits over other systems.

In our research, we have conducted a systematization of the cross-context communication ecosystem and an evaluation of over 25 systems. The evaluation of existing systems gave the following insights our research. First, a small number of systems supports communication with worker contexts and even a smaller number of systems enable communication with both window and worker contexts. Second, a small number of evaluated systems support high-level communication models like remote procedure call and publish-subscribe. Third, a small number of systems provide more than one communication model. Fourth, context discovery is not addressed by any of the evaluated systems. Lastly, although security features of cross-communication systems have been the most researched, the authorization aspect of security is still significantly underdeveloped.

We believe that future cross-context communication systems should be guided by the principle of economy of liabilities [7] and hide the complexity of cross-context communication by providing high-level functionalities. In this paper we present Pmrpc, a novel cross-context communication system which addresses the stated issues of existing systems. Pmrpc is a JavaScript library based on the standardized HTML5 primitives which provide secure message-based communication between Web application contexts. The architecture of Pmrpc provides and unifies three different types of communication models under the same interface - message-based communication, remote procedure call (RPC) and publish-subscribe. Furthermore, the same interface enables communication with both window-type and worker-type contexts. Lastly, the system enables dynamic discovery of Web application contexts and specification and a whitelist-based access control model of authorization. We evaluate the system by comparing the execution times of achieving specific application goals when using Pmrpc to those of HTML5 primitives.

The rest of the paper is organized as follows. In section 2 we outline the related work of our research while in section 3 we present the architecture of the Pmrpc cross-context communication system. In section 4 we present the evaluation of the system while Section 5 concludes the paper with directions for future work.

## II. BACKGROUND

Browsers manage the execution of each Web application using semi-isolated environments called browser execution contexts, sometimes also called script contexts [8] [9]. Web applications may be built from many parts, each part executing in its own context. There are two main types of execution contexts: *GUI-based browsing contexts* [8], such as windows and iframes, and GUI-less thread-like worker contexts [4], which are further divided into shared and dedicated worker contexts. We use the term *cross-context communication* to define the process of transferring data across context boundary.

Many systems have been developed to enable cross-context communication. Early systems, such as FIM [10] and window.name [11], were based on browser mechanisms and quirks for bypassing the Same Origin Policy (SOP) [12] [13]. SOP is a browser security policy which almost completely restricts Web applications executing in a browser to communication between contexts on the same trust domain, also called an *origin*. Only the recent HTML5 [8] and Web Workers specifications [4] have defined standard browser APIs for secure massage-based cross-origin cross-context communication. Recent client-side frameworks, such as Google Closure [14], easyXDM [15] and jsChannel [16], are built upon these primitives to provide high-level communication models such as RPC, backward compatibility with older browsers which do not support the HTML5 group of standards, cross-browser support and other advanced features.

As a part of our research, we have conducted a broad analysis of over 25 existing cross-context communication systems. The analysis has highlited the following directions for future research of cross-context communication. First, in order to reduce application-level complexity of multi-context applications that require cross-context communication, cross-context communication systems should unify both window and worker context comunnication as well as more than one communication model behind a uniform interface. This makes it possible to achieve required cross-context interaction using only a single system instead of several, consequently reducing overall complexity and increasing performance due to fewer network requests for fetching systems' libraries. Furthermore, cross-context communication systems should support high-level communication models, like RPC. These communication models are often preferable over simple message-oriented models since they require a smaller code overhead for achieving application goals. Finally, context discovery, reliable communication and authorization aspects of security should be supported to enable the use of the systems in dynamic Web applications created as mash-ups of contexts from different trust domains.

The Pmrpc system presented in this paper uses the HTML5 and WebWorkers standard primitives as the transport mechanism and JSON-RPC [17] as the communication protocol. HTML5 and WebWorker primitives are based on the postMessage API for sending messages to remote contexts and the onMessage event for receiving messages in those remote contexts. JSON-RPC is a transport-agnostic and stateless RPC protocol which uses JSON as the message data format. The protocol defines two message types, a request and response. The request contains a request identifier, remote method name and parameters, while the response contains the identifier of the request message, result of the invocation and an error object.

## III. PMRPC CROSS-CONTEXT COMMUNICATION SYSTEM

Pmrpc [18] is a JavaScript library for cross-context communication. The purpose of the Pmrpc project is to research interesting directions of cross-context communication. Specifically, we explore the challenges in and advantages of unifying different communication models, such as RPC and publish-subscribe, and communication between contexts of any type, such as window and worker types, in a single system. Unification implies that the same methods of the system's interface may be used to achieve cross-context communication between contexts of any type using any supported communication model. The main benefit of unification is that a single cross-context communication system may be used in place of many systems which support only specific communication models or context types. The main challenge for unification of all context types are the differences in interfaces of native browser primitives for communicating with these context types. The main challenge for unification of different communication models are the implementation of higher-level models using native browser message-oriented primitives and the differences in context naming used in different models.

The Pmrpc system may be classified as follows. Pmrpc is a client-side framework that doesn’t use external components for establishing communication nor for transferring data. The system supports communication between contexts on different origins and its usage is not restricted to any specific Web applications. The standard HTML5 and WebWorker postMessage primitives are used as a transport system and JSON-RPC is used as the communication protocol. Consequently, Pmrpc may be used on browsers that implement the HTML5 and WebWorker specifications. The system unifies communication between contexts of any type and three communication models: message-based communication, RPC and publish-subscribe.Two types of naming may be used with Pmrpc: context object references combined with custom string names for message-based and RPC communication, and custom string channel names for publish subscribe communication. Pmrpc supports discovery of procedures and channels within a single Web application. Reliability of communication and fault tolerance is based on a retry mechanism. For message-based and RPC communication, Pmrpc supports unicast and multicast communication, while broadcast communication may be achieved using the discovery

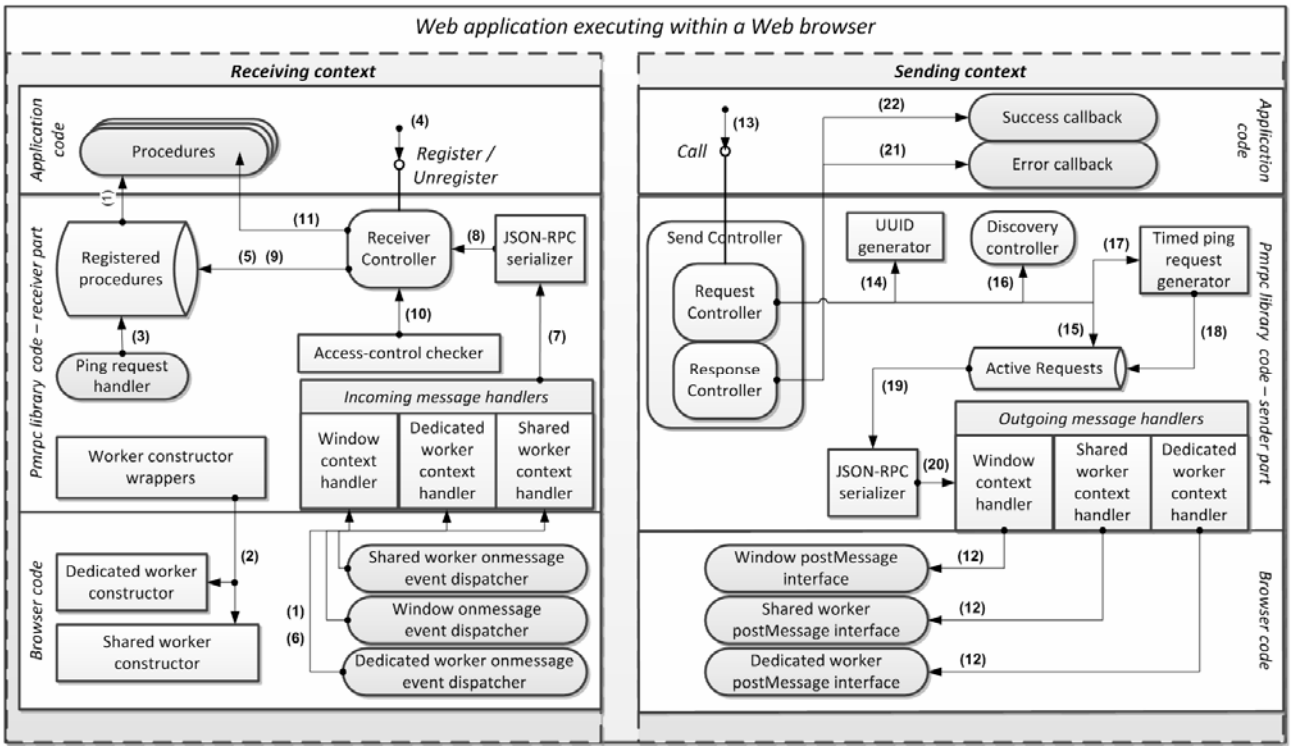


Figure 1. Architecture of the Pmrpc cross-context communication system

mechanism. The confidentiality, integrity and authentication security features are inherited from the HTML5 and WebWorkers primitives while Pmrpc provides advanced authorization through a whitelist access control list mechanism. In the following sections we first give an outside view of Pmrpc by describing its programming interface (API) and then give an inside view by describing its architecture

### *A. Pmrpc programming interface*

When loaded into a browser execution context, the Pmrpc library exposes a generic API based on a RPC model. This API, depending on parameters passed to the methods, may also be used for message-based and publish-subscribe communication. The API consists of the following four methods.

**Register(p, name, ?acl)** – exposes a procedure *p*, defined in the current context, under the name *name* so that it may be called from remote contexts. Procedure names are specified using custom character strings with no restriction on the format of the string. Furthermore, an optional access control list *acl* may be passed, specifying a whitelist of remote context origins which may invoke the procedure. In a publish-subscribe model, this method is used as a subscription to a channel with name *name*.

**Unregister(name)** – removes a previously exposed procedure with name *name*.

**Call(name, ?dest, ?args, ?acl, ?retries, ?timeout, ?onSucc, ?onErr)** – invokes a remote procedure exposed with name *name* in every context defined in the *dest* array. Contexts in the *dest* array are defined by a window or worker object reference. If the destination array is not specified, the named procedure will be called on every discoverable context, which simulates publishing a message to a channel named *name* in a publish-subscribe context. Arguments for invoking the remote procedure may be passed through the *args* array. Optionally, an access control list *acl* may be passed, specifying a whitelist of origins. If the origin of a destination context specified in *dest* is not listed in *acl*, the call will not be made to that context. If specified, the *retries* parameter defines how many times will Pmrpc attempt to call a specified destination context before giving up, and *timeout* defines how many milliseconds should the system wait for a response before concluding that the called procedure is not available. The argument passed as the optional function parameter *onSucc* will be invoked in case a call was successful and a result was received while *onErr* will be invoked in case the remote call has failed for any reason. If neither of the function parameters were passed arguments, the Pmrpc call method simulates a one-way message without a response.

**Discover(dest, cb, ?origin, ?name)** – discovers Pmrpc procedures registered on contexts specified in the optional array *dest*. If the *dest* array is not specified, Pmrpc procedures are discovered on all directly reachable contexts. In case this method is called from within a window context, all window contexts in the Web application and directly nested worker context will be found. However, if this method is called from within a worker context, only the parent context of the worker and directly nested worker contexts will be found. Optionally, *origin* and *name* lists of regular expression strings may be specified to filter discovered methods based on the origin of the registering context and the name of the procedure.

The *cb* callback function will be invoked with the list of discovered procedures, specifying for each procedure the name of the procedure, the access control list defining authorization for the procedure, the context object of the registering context, and the origin of the context.

### *B. Pmrpc architecture*

The main goal when designing the Pmrpc system was to achieve a single and uniform interface for different context types and different communication models. The main challenge was wrapping differences of using the native HTML5 and WebWorker primitives for different combinations of sender and receiver context types; both the sender and receiver context may be either a window, dedicated worker or shared worker context.

Fig 1. shows the architecture of the Pmrpc system in a Web application environment consisting of two contexts: a context sending data and a context receiving data. Both contexts contain three levels of code: application-level code, the Pmrpc system and browser code exposed through browser APIs. The Pmrpc system is logically divided into the receiver part and sender part. Modules shown on the right hand side are responsible for sending a RPC request to a remote context and receiving the response and comprise the receiver part. Modules shown on the left hand side are responsible for registering procedures and responding to incoming remote calls and comprise the sender part. Because both parts of the system are always present, each context may be used to both send and receive Pmrpc calls.

**Receiving logic** – when loaded into a browser execution context, the Pmrpc library needs to ensure that all incoming Pmrpc messages will be delivered to the Receiver Controller module. First, for cases in which the context hosting the system is a window context receiving messages from another window context, or a worker context receiving messages from the parent context, Pmrpc attaches event handlers for incoming Pmrpc messages to the browser onMessage event dispatching APIs (1). Second, for cases in which the context hosting the system is a parent window or parent worker context receiving messages from nested workers, the library wraps the dedicated and shared worker constructors (2) to automatically attach Pmrpc event handlers for the onMessage event of created workers. The system then registers a special internal Ping method (3) for responding to requests which are sent by the sending context to test if a procedure is exposed remotely and available. Using the **register** method of the Pmrpc system, application-level logic registers a procedure (4) and the procedure name, a reference to its implementation function and access control rights are stored in the Registered procedures storage object (5). When a remote procedure call request is received by the Incoming message handlers (6), the message is deserialized by the JSON-RPC serializer (7) and passed to the Receiver Controller (8). The Receiver Controller fetches the information for the requested procedure from the Registered Procedures storage (9). If the Access Control checker permits the invocation of the procedure based on the authorization rules of the procedure and the origin of the sending context (10), the Receiver Controller invokes the procedure (11), receives the result and serializes it into a JSON-RPC response message. The response is then sent back to the sending context by internally invoking the Pmrpc call message. Lastly, if the **unregister** method is called by application-level logic (4), the name and the reference to the procedure are removed from the Registered procedures storage object (5).

**Sending logic** - when loaded into a browser execution context, the Pmrpc system creates function wrappers around postMessage APIs for different types of possible destination contexts (12). The system then waits for invocations of the call method from application-level logic. For each invocation of the **call** method (13), the Request Controller generates an identifier for the request using the UUID generator (14) and creates an entry in the Active Requests storage (15). If the invocation of the call method didn't specify the array of destination contexts, the Pmrpc sending logic invokes the Discovery controller (16) to dynamically discover which contexts implement the named remote procedure. Furthermore, before sending the request to each destination context, the sending context checks if the procedure to be called is available. This check is performed by starting a timer (17) which periodically generates a remote procedure call request for the Ping procedure (18) by internally invoking the call method. The ping request is serialized into a JSON-RPC request message (19) and sent to the destination using postMessage wrappers (20). If a response isn't received in the timeout period specified in the call method invocation, the ping request is repeated. If the ping request fails more than the number of times specified in the retries parameter of the call method, the request is considered failed and the error callback is invoked (21). If a ping request succeeds, the pinging process is stopped and the real request is sent to the destination. Upon reception of the response, the success callback is invoked (22).

**Discovery logic** - Pmrpc implements discovery logic as a special case of receiving and sending logic. Upon loading, the Pmrpc system registers a *Discover Registered Procedures* procedure which may be called from remote contexts. When called remotely, the procedure returns the current state of the Registered Procedures storage object and the origin of the context that contains it. Upon invoking the Pmrpc **discover** method, the discovery logic calls the *Discover Registered Procedures* procedure on each specified context to obtain all registered procedures. The obtained lists of procedures are then filtered by regular expression filters passed in the invocation of the discover method. If no contexts were specified in the call of the discover method, the method first discovers all reachable contexts. Window contexts are discovered by traversing the window context tree starting from the top window of the application [8]. This process is implemented by recursively visiting elements of the window.frames array. The window.frames array is

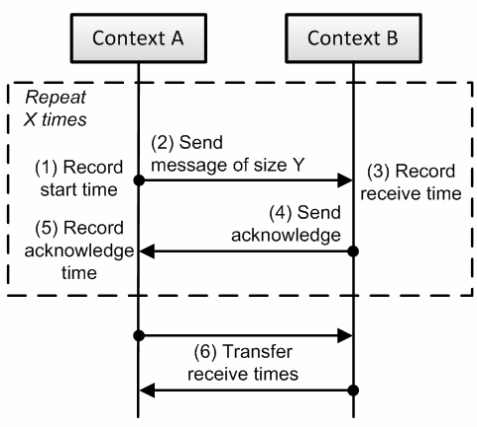


Figure 2. Evaluation scenario sequence diagram

exposed in every window context, may be accessed from remote contexts on different domains and contains a reference to every directly nested window context. Worker contexts are discovered by requesting a list of created workers from the pmrpc receiving logic. Since the receiving logic wraps both shared and dedicated worker constructors for receiving message, it additionally keeps an internal list of all instantiated workers.

## IV. EVALUATION

In order to evaluate the run-time properties of the Pmrpc system, we measure and compare its performance to that of the browser native postMessage mechanism, in a predefined scenario. Fig. 2 shows the sequence diagram describing the evaluation scenario. In the scenario, two browser window contexts exchange messages, with context A as the sender and context B as the receiver. The actual message exchange is a sequential repetition of the following interactions: context A sends a message to context B, context B receives the message and finally context B sends an acknowledgment back to context A. Absolute times are noted upon sending the initial message (1), context B receiving the message (3) and context A receiving the acknowledgement from context B (5). Finally, after all repetitions have finished, context A collects the noted times from context B to perform evaluation calculations (6). Two parameters are variable in the scenario; the number of repetitions, denoted by X in Fig. 2, and message size, denoted by Y.

The same scenario was used for evaluating both Pmrpc and postMessage performance in two experiments described below. The evaluation was performd on a machine equipped with a Intel(R) Core(TM)2 Duo 2.26GHz processor with 4GB of RAM using the Linux operating system. We used the latest stable build of the Chromium Web browser (v8.0) and Firefox Web browser (v3.6.12). However, we show measurements only for the Chromium due to Firefox crashing for most test cases, for both postMessage and Pmrpc systems alike.

In the first experiment we varied the number of repetitions *X* with a constant message size *Y* of 1KB. Fig. 3 a) shows the results of the first experiment, with the horizontal axis denoting the number of repetitions, from 1000 to 50000, and the vertical axis denoting time duration in milliseconds. For Pmrpc and postMessage each, the graph plots two measurements. First, plots “Pmrpc one way” and “PostMessage one way” show the average time spent on sending a message from context A to context B, as the difference of times recorded at points (1) and (3) in Figure 2. Second, plots “Pmrpc ack included” and “PostMessage ack included” show the average time spent on sending a message from context to context B and receiving an acknowledgment, as the difference of times recorded at points (1) and (5) in Fig. 2 The results shown in Fig. 3 show that the Pmrpc system is 5 to 6 times slower than the postMessage mechanism.

In the second experiment we varied the message size Y with a constant number of repetitions X of 2000. Fig. 3 b) shows the results for the second experiment, with the horizontal axis denoting the message size, from 1B to 100KB, and the vertical axis denoting time duration in milliseconds. The meaning of graph plots are the same as in the previous experiment. The results show that the Chromium browser has no noticeable slowdown until 10 KB message sizes are reached. Furthermore, at 100KB there is a noticeable reduction in performance for both postMessage and Pmrpc. Lastly, it can be noticed that Pmrpc can handle the same amount of data as postMessage, again being between 5 and 6 times slower.

The reduced performance of Pmrpc when compared to postMessage was expected due to the added features of Pmrpc. First, the most significant reduction in performance is a result of the pinging process for ensuring reliability. Since every invocation of the Pmrpc call method results in at least one ping request to the destination procedure and at least one response to that request, the number of postMessage messages exchanged is increased by at least two messages, i.e. at least 100%. Second, unlike the postMessage mechanism, Pmrpc wraps messages in a JSON-RPC request and response thus increasing the number of bytes to be transferred and decreasing performance. Third, Pmrpc performs several time consuming tasks for each remote call, such as serializing JSON-RPC messages and checking access control rights using regular expressions. However, the reduced performance is of little significance since both systems still operate within the 1ms order of magnitude. Additionally, Pmrpc hides the complexity of implementing the proposed features, including access control features. In result, Pmrpc is more in line with the principles of the Economy of liabilities [7] which states that a system should minimize the liability that the user undertakes to ensure application security.

## V. CONCLUSION

Similar to multi-process desktop applications executing on operating systems, modern Web applications are built from many browser execution contexts. Therefore, a fundamental requirement for Web browsers is adequate support for cross-context communication. However, existing cross-context communication systems are trailing behind Web

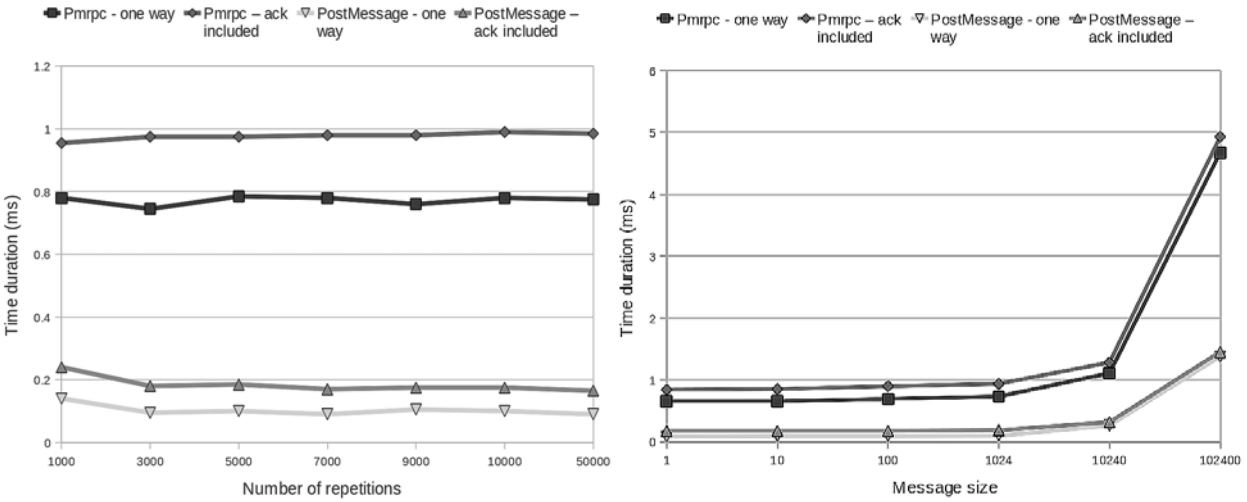


Figure 3. a) (left) Results for Experiment 1: measurements of the evaluation scenario for varied number of repetitions and constant message size (1KB) b) (right) Results for Experiment 2: measurements of the evaluation scenario for varied message sizes and constant number of repetitions (2000).

application requirements. We believe that future cross-context communication systems should hide the complexity of cross-context communication by providing high-level functionalities and unifying them under the same interface.

In this paper we present Pmrpc, a cross-context communication systems which unifies three different communication models for communication with all types of browser execution contexts under the same interface. Furthermore, Pmrpc implements cross-context communication reliability and authorization, context discovery and provides these features through the unified interface. Although Pmrpc request-response calls are several times slower than a request-response cycle using native browser primitives, this result is expected due to the features that Pmrpc provides. More importantly, the execution time of a single Pmrpc request-response call is in the millisecond range of native primitives which supports the usage of the library in real world applications. Lastly, Pmrpc reduces the complexity of achieving application goals in multi-context Web applications when compared to implementing the same goals using native browser primitives.

Pmrpc is a free and open-source Apache 2.0 licensed project, and has been so for more than a year. In that period, the project has inspired the development of several other systems, including easyXDM [15] and jsChannel [16]. Our work on Pmrpc has given us many insights and opened many possible areas for future work. For example, the jsChannel system improves the performance of reliable communication by checking the availability of a specific remote procedure only on the first call, while Pmrpc performs this check on every call. Furthermore, we have started working on extending Pmrpc for cross-browser cross-context communication which will enable calling procedures across Web application and browser boundaries.

## REFERENCES


[1] Fraternali, P., Rossi, G., and Sánchez-Figueroa, F. 2010. Rich Internet Applications. In IEEE Internet Computing. 14, 3 (May-June, 2010). 9-12.

[2] Yu, J., Benatallah, B., Casati, F., and Daniel, F., 2008. Understanding Mashup Development. In IEEE Internet computing. 12, 5 (Sept.-Oct. 2008), 44-52

[3] Srbljic, S., Skvorc, D., and Skrobo, D., 2009. Widget-Oriented Consumer Programming. In AUTOMATIKA: Journal for Control, Measurement, Electronics, Computing and Communications. 50, 3-4 (Dec. 2009). 252-264

[4] Hickson, I. Ed., Web Workers. W3C draft (accessed on Sept. 8, 2010). http://dev.w3.org/html5/workers/

[5] Reynolds, F. 2009. Web 2.0–In Your Hand. In IEEE Pervasive Computing. 8, 1 (Jan.-Mar. 2009), 86-88

[6] Silberschatz, A., Galvin, P. B., and Gagne G. 2008. Operating System Concepts; 8th Edition. Wiley, 2008.

[7] Hanna, S., Shin, R., Akhawe, D., Saxena, P., Boehm, A., and Song, D. 2010. The Emperor's New API: On the (In)Secure Usage of New Client Side Primitives. IEEE Web 2.0 Security and Privacy Workshop (May, 2010).

[8] Hickson, I. Ed., HTML5:A vocabulary and associated APIs for HTML and XHTML. W3C draft (accessed on Sept. 8, 2010). http://www.w3.org/TR/html5/

[9] Reis, C., Gribble, S. D., and Levy, H. M. 2007. Architectural principles for safe web programs. Sixth Workshop on Hot Topics in Networks (Atlanta, Georgia, November, 2007).

[10] Jackson, C., and Wang, H. J. 2007. Subspace: Secure CrossDomain Communication for Web Mashups. 16th international conference on World Wide Web (Banff, Alberta, Canada, 2007), 611-620.

[11] Zyp K. 2008. window.name transport (accessed on Sept. 8, 2010). http://www.sitepen.com/blog/2008/07/22/windowname-transport/

[12] Zawelski, M. Browser security handbook. (accessed on Sept. 8, 2010). http://code.google.com/p/browsersec/wiki/Main

[13] Barth, A., Jackson, C., and Hickson, I. The Web Origin Concept. IETF Internet-Draft (accessed on Sept. 8, 2010). http://tools.ietf.org/id/draft-abarth-origin

[14] Google Closure Library (accessed on Sept. 8, 2010). http://code.google.com/p/closure-library/

[15] Kinsey, Ø. S. easyXDM framework (accessed on Sept. 8, 2010). http://easyxdm.net/

[16] jsChannel (accessed on Dec. 29, 2010). https://github.com/mozilla/jschannel

[17] JSON-RPC 2.0 Specification. (accessed on Dec. 29, 2010). http://groups.google.com/group/json-rpc/web/json-rpc-2-0

[18] Zuzak, I., and Ivankovic, M. Pmrpc library. (accessed on Dec. 29, 2010). http://code.google.com/p/pmrpc/