Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 150 (2019) 61–68

Procedia Computer Science

www.elsevier.com/locate/procedia

13th International Symposium "Intelligent Systems" (INTELS'18)

The Design Principles of Intelligent Load Balancing for Scalable WebSocket Services Used with Grid Computing

V.A. Alexeev, P.V. Domashnev, T.V. Lavrukhina, O.A. Nazarkin*

Lipetsk State Technical University, Moskovskaya str., 30, Lipetsk 398055, Russia

Abstract

Large-scale grid compute projects may use computational capabilities of web browser sessions on numerous client devices to run distributed data processing algorithms. WebSocket protocol is the basis of duplex communications between browser worker nodes. Publish/Subscribe pattern defines communication scheme for client-server messaging. The downside of WebSocket-based Publish/Subscribe implementations is the stateful nature of long-living network connections. This aspect makes WebSocket scaling and load balancing a challenging task. In present work, we state that the intelligent balancer needs to consider different behavior types of client connections. We propose the negotiation-like mechanism between client and the load balancing system about the target server to set the client on. Our construction principles of scalable multi-server infrastructure with effective inter-server communications intensively use Redis instances, because Redis is a mature, well-tested, high performance system, rich of functions.

Keywords: browser-based grid computing; WebSocket protocol; horizontal scaling; load balancing; Publish/Subscribe; Redis applications.

1. Introduction

In this paper, the general scheme of horizontal scaling of WebSocket-based web services is proposed. We think that the importance of WebSocket scaling and load balancing is crucial for the tasks of large-scale browser-oriented grid computing systems, because of the unprecedented convenience of WebSocket for message passing between

* Corresponding author.
E-mail address: nazarkino@mail.ru

10.1016/j.procs.2019.02.014

numerous web nodes. The stateful character of WebSocket long-living connections needs special approach to construct intelligent load-balanced server platform.

1.1. The potential of browser-based grid computing

Using in-browser JavaScript context to obtain distributed computational resources for data processing tasks becomes an important paradigm for large-scale grid computing [1,3]. Active web sessions can form a globally distributed computational grid, with millions of processors. Targeting compute code to browsers is especially efficient to build large-scale volunteer computing projects [2]. Any personal computer, used actively for work and entertainment, has up-to-date web browser. User needs no additional software installation to join volunteer computing grid as worker node, if the grid runs browser-oriented worker modules. The simpler joining procedure makes better possibility to involve web surfer into grid project.

Modern browsers, powered by high performance JavaScript engines, such as Chrome V8, Mozilla SpiderMonkey, and others, provide rich runtime environment that is very often underutilized as a target computational platform. Advent of WebAssembly [4] brings possibility to run code in web browser at near-native speed. Web Worker [5] API, well supported among browsers, allows creating parallel JavaScript contexts. Web workers are isolated from host HTML document, but they are capable to exchange messages with main JavaScript thread, and they can make HTTP or WebSocket connections for network I/O to communicate freely with outer world.

1.2. Publish/Subscribe as the main communication pattern for browser-based grid computing

Web systems mostly use client-server architecture. Web browsers are client nodes, so JavaScript code in Worker context can communicate with any web server, but not with other remote client nodes directly. Obviously, this is not the shortest communication path between two computational nodes. A distributed algorithm that needs intensive inter-node messaging is forced to transfer all messages through the server, thus making high load on it.

The brand new WebRTC [6] API brings p2p (peer-to-peer) capabilities to browser networking. However, the downside of WebRTC is NAT traversal issues, making initiation of p2p-sessions somewhat unstable, in general. Despite lower latency and logical attractiveness of p2p topologies, we think that WebRTC is not the option to choose for web grid computing. WebRTC best suits one-to-one communication pattern, not many-to-many. With WebRTC there is no practical possibility of casting messages directly from one node to the large set of other nodes, because browser context cannot establish and hold thousands of p2p sessions. In web-oriented grids, the set of worker nodes varies dynamically, as web sessions in global perspective connect and disconnect in chaotic manner. Therefore, a worker node in the web grid does not know exactly its remote partners.

On the contrary, centralized client-server architecture suits well to exchange messages within a large set of nodes. Publish/Subscribe model [7,8] is proper abstraction to define communication scheme for client-server messaging, where server part is “message broker”, or “message bus”. Every client node can subscribe to some channels and publish to others. By decoupling sender and receiver nodes, centralized Publish/Subscribe model easily solves the problem of casting messages to undefined target set. Sender (publisher) does not know identifiers of message receivers (subscribers), it knows only the common channel identifier to publish into, and the broker is responsible to notify all subscribers to this channel. There is only one connection between any worker node and grid system as a whole – it is the connection to central server. Both workers and work dispatchers use this central message broker to cast tasks, interchange temporary data while processing, and then gather results. Publish/Subscribe is good for broadcasting, but it is also applicable to one-to-one or many-to-one message delivery. In these cases, the addressee must subscribe to a channel with unique name, and all interested nodes in a grid have to know this channel name.

1.3. WebSocket as the main communication protocol for browser-based grid computing

Publish/Subscribe model needs long-living duplex connections between client and server, due to asynchronous character of publish events. Subscribers should wait notifications for arbitrary periods, and server has to initiate published data transfer to a subscribed client (push mode). Before WebSocket [9,10] it was long polling technique to emulate server-initiated pushes, with lots of drawbacks of such a solution. Now WebSocket is standardized, lightweight protocol, well supported in all variety of modern JavaScript engines.

WebSocket protocol itself does not define any elements of Publish/Subscribe model; it operates with "send" commands and "onmessage" handlers, called back on asynchronous data reception. However, many popular WebSocket client libraries use high-level abstractions in the form of Publish/Subscribe, where "send" is publish, and "onmessage" handler locally dispatches notifications to the proper channel listeners. In this scheme, message becomes a structured data block, containing metadata (at least, channel name) and a payload.

The downside of WebSocket is the stateful nature of long-living connections. WebSocket server must maintain control entry for any open connection. When large number of clients connect, server may become out of memory resources, even if most of connections just passively wait for data transfer. Moreover, if high-level WebSocket abstraction uses Publish/Subscribe model, the server side must keep all subscriptions of every client connection.

2. Intelligent load balancing for WebSocket services

2.1. Prerequisites: sticky sessions as trade-off automatic solution for WebSocket services scaling

High-performance grid computing implies many thousands, or even millions simultaneously working nodes. Browser-based worker nodes use centralized Publish/Subscribe service built over WebSocket. At present, there are little means to construct effective web-oriented Publish/Subscribe over stateless protocol, such as HTTP. This condition leads to difficulties with scaling and load balancing of WebSocket-based services. The aim of horizontal scaling is to handle more clients connections than every single server is capable to maintain. The other benefit is redundancy – if one server fails, others are available.

Scaling is generally a server-side "problem". Clients do not see the server pool, they always connect to the single service endpoint. The server-side entry (Load Balancer) gets these connections and forwards requests to other servers from dedicated pool. The same is for responses – they are transferred back to clients through the Balancer (Fig. 1).

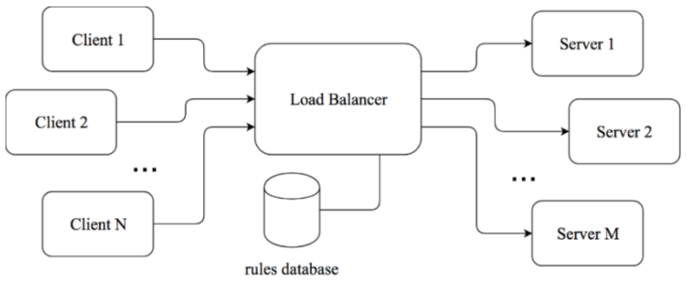

Fig. 1. General load balancing scheme for web systems scaling.

The scheme works fine with short-lived stateless HTTP requests, as balancer does not need to keep connections in memory; it closes them after response is returned to client. Request from any client can be forwarded to any server, depending on current load distribution. This is not the case with WebSocket connections, as they are open TCP sockets. Any traffic from, say, Client #1 must be forwarded to the server which was first selected by the balancer to serve on this client's connection (say, Server #2), or else the TCP tunnel through balancing proxy will be destroyed. As a consequence, the long-lived WebSocket session of Client #1 "sticks" to the Server #2. "Sticky sessions" persist for any client-server pair (session affinity). Industrial load balancers (e.g., Nginx [11,12]) support sticky sessions and can be used to scale WebSocket. When only connections count is considered, sticky sessions perform well, and scaling is just straightforward: system needs more servers to handle more clients.

The pitfall is that connections make unequal impact to the server-side. Some clients connect and wait, doing nothing, with zero traffic. Some clients are active, continuously transmitting lots of data. The main thesis we state here is that there is no "general type" of real-time duplex WebSocket connection. They all are unique in behavior.

Moreover, there is no “general type” of WebSocket-based service – in present work, we consider Publish/Subscribe brokers with real-time notifications, as this model is in focus of our tasks of compute grid inter-node communications. Other applications may require different scaling logic. In addition, we point out that *scaling* and *load balancing* are interleaved but different paradigms in the context of WebSocket. Scaling targets to handle new clients, while load balancing tries to cope with existing clients, during their long-living activity.

2.2. Classification system for types of connection behavior and server conditions in Publish/Subscribe communication

To cover the most wide set of client behavior types we construct an orthogonal space with the following axes: 1) SP – temporal subscriptions profile; 2) TP – temporal traffic profile; 3) LE – life term expectation; 4) RO – reconnection overhead; 5) MS – maximum subscriptions expectation, 6) MT – maximum traffic expectation.

The SP factor defines the subscriptions count as function of time. The TP factor defines traffic intensity as function of time. The LE factor is related to the expected duration of client session. The RO factor is related to cost of client’s reconnection, if forced by server, in the context of client’s application tasks. All factors are independent, so we can construct a 6D-tuple to characterize arbitrary connection behavior type. It is impossible (and useless) to introduce any quantitative precision in description of the stated factors; the reasonable approach is to qualify their values as elements of discrete sets, such as {zero, low, medium, high} for LE, RO, MS, MT, or {constant, periodic, bursts} for SP, TP. For example, the tuple {SP=constant, TP=bursts, LE=high, RO=low, MS=low, MT=high} describes a long-lived client with a fixed small number of subscriptions, capable to produce or consume high traffic in bursts; this client can easily survive reconnects, if needed.

In addition, we need an approach to dynamically measure overall current condition of every server in the pool. We use following independent axes to construct orthogonal space: 1) CC – connections count; 2) PU – CPU usage; 3) MU – RAM usage; 4) BU – network bandwidth usage. CC factor is integer, PU, MU, BU are percent. The 4D-tuple describes current server state, and series of periodically registered tuples may form retrospective history useful for decision-making.

Load balancing algorithm should use this structured knowledge. Above noticed “server-side problem” of load balancing turns out a problem for clients. In case of improper declared strategy, proxy appoints client to the wrong server. For example, the simple sticky sessions load balancer chooses end server with few present connections, all of them lazy, transferring near zero traffic – good to land newcomer on. At the next moment – because high-level application context changed by the joined new member – all the clients on the chosen server become highly active, and will surely suffocate, or even become forcibly disconnected, if server runs out of resources.

The algorithm of effective WebSocket load balancing should face the unpredictability of clients. The above-defined 6D-tuples of clients and 4D-tuples of servers (of course, both may be extended with more elements) can help an intelligent load balancer to choose the right strategy for a new connection in existing server-side state.

2.3. The two-stage connection procedure for WebSocket-based Publish/Subscribe service

WebSocket load balancing is mutual problem of client and server. Without a hint from client, it is hardly possible to get 6D-tuple knowledge on the server side. A reasonable solution is to lift load balancing to the application level. It means that client application logic will be aware of server-side scaling. The connection procedure becomes two-stage (Fig. 2).

At first, client requests special Orchestrator service to recommend a server endpoint to connect to, providing connection behavior hint (6D-tuple) to the server side. Orchestrator is not a Load Balancer, generally. It is the decision-making and advising service. Orchestrator does not hold TCP tunnel from client to server. Instead, client directly connects to the server of choice. End servers from the pool periodically report their state (4D-tuples) to the Orchestrator, so it has overall and detailed load distribution at hand. The heuristic algorithm, which can be very complex (artificial neural networks are implied candidates), responds to the client request by returning “best fit” server endpoint, in present conditions. Client then connects directly to the recommended server. The response should contain special connection token, which the end server checks on connection – the token is needed to filter out clients who ignore the Orchestrator. Token has relatively short time to live, and then becomes invalid. This mechanism makes client appointments always up-to-date, forcing them to connect as fast as possible while dynamic load conditions have not changed much since the moment of appointment.

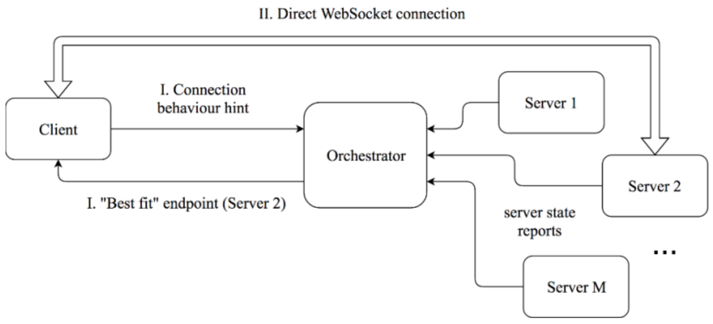
II. Direct WebSocket connection
I. Connection behaviour hint
Client
Orchestrator
I. "Best fit" endpoint (Server 2)
Server 1
Server 2
server state reports
...
Server M

Fig. 2. Application level load balancing for WebSocket scaling.

Orchestrator service as a separate management stage brings additional degrees of freedom to clients. The Orchestrator may return “behavior correction” hints – its response may contain several recommended servers, best fitting different client behavior types. This function is helpful for self-organizing compute grids, where new joined grid node can choose tasks according to its processing and communication capabilities; node may get, for example, large packets for infrequent upload/download (bursts-type communication behavior), or small packets with short processing time and stable network I/O flow. Therefore, the feedback from Orchestrator helps to expand intelligent client behavior patterns.

2.4. Using Redis to organize Publish/Subscribe bus

Publish/Subscribe communication pattern implies that any Publisher client can transfer data to any Subscriber client through the central server-side broker service. When the WebSocket service is distributed among separate servers, it needs inter-server connections. One can view the interconnection as a common message bus shared by all servers of the pool. The bus is actually a process that listens for messages and transfers them between servers. There is a convenient industrial solution for this task. It is Redis [13], in-memory data store, often used as message broker. Redis supports Publish/Subscribe “out of the box”, so WebSocket server can delegate client subscriptions and publish commands to Redis (Fig. 3).

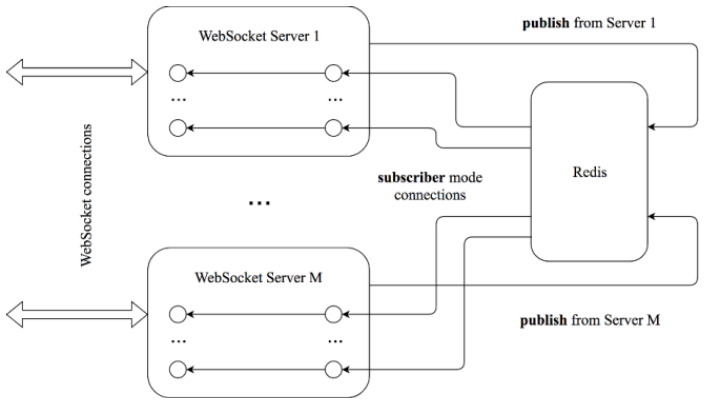

Fig. 3. Redis instance as Publish/Subscribe broker.

Using Redis as Publish/Subscribe broker has many benefits. First, Redis is mature, heavily tested high performance system, it implements very efficient in-memory data structures, accessed with fine-tuned algorithms. Second, Redis itself has scaling potential (Redis Cluster [14] solution). Third, delegation simplifies WebSocket server algorithm, thus making its implementation lightweight, with much less memory consumption – as a sequence, the server can maintain more client connections and transmit data faster.

Each WebSocket server opens only one Redis connection for publish commands (it is shared by all WebSocket clients), and many connections (equal to clients number) to subscribe to channels. There is one-to-one correspondence between WebSocket connections and Redis subscriber connections, which are in special mode – they can listen for notifications on channels events, or issue subscribe/unsubscribe commands. When Redis notifies on subscriber connection, WebSocket server simply replicates received data to the corresponding WebSocket connection.

2.5. General scheme for WebSocket scaling with multiple Redis instances

In production, WebSocket services have to work with more simultaneous clients than single Redis-based Publish/Subscribe broker instance can handle. As above stated, there is scalable Redis Cluster solution that targets the most heavy cases. Redis Cluster allows publishing data on any cluster node, and corresponding subscribers would be notified on any node. However, this solution is not general, and it has limitations and drawbacks. Here we propose the scaling scheme with separate Redis instances, not a Cluster. Our main goal is to optimize publish commands, because Redis Cluster (at least present) simply broadcasts any publish command to every node in the cluster, therefore, it may lead to significant overhead. In addition, we aim to construct scalable cluster-like solution that may, in its turn, aggregate Redis Clusters, if needed – it would open the next level of scaling.

We propose general scheme for WebSocket-based Publish/Subscribe services scaling (Fig. 4). It uses multiple WebSocket servers (WS) paired with Redis instances as Publish/Subscribe handlers (PSH). The above-described (section 2.3) two-stage connection procedure is assumed. The Orchestrator component is accessed by http(s) through REST API, therefore, it can be load balanced with common scaling solutions, like Nginx. As mentioned in sections 2.2 – 2.3, the intelligent Orchestrator logic is rather complex, so each client request should be forwarded to one of the large set of equal processing blocks, to keep the response time short while running the decision making algorithms (heuristic, or neural). All processing blocks of the Orchestrator are connected to Redis instance LSR – Load Statistics Registry (it can be Redis Cluster, if needed). Every WS in the pool periodically writes its current load state to the LSR, so Orchestrator has enough data to choose an appropriate WS for the client request.

The Orchestrator returns the WS identifier coupled with the short-living unique token (“access ticket”), also stored in LSR. Client must use the token to connect to the selected WS, avoiding delays (or token becomes invalid). Redis supports TTL (“time to live”) for the keys, which is very convenient – a key is automatically deleted by Redis after the specified TTL ends. On client connection, WS checks token key in LSR, and in case token is not fresh (the corresponding key is not found, because it is already automatically deleted), WS denies the connection. The same is for tokens not returned by Orchestrator (“fake” tokens). WS must delete the token from LSR on successful connection.

Each WS can maintain client subscriptions on arbitrary channels, using one-to-one WebSocket-to-Redis connections mapping. WS delegates subscriptions to the corresponding PSH. All subscribe and unsubscribe commands are forwarded to PSH, and all notifications from PSH (channel events, with published data) are transmitted back to WebSocket connections. On WebSocket client disconnect, WS must close the paired Redis connection (PSH itself will do the rest, destroying the client’s subscriptions list).

Publish commands need special forwarding to the PSH instances. The simple algorithm just broadcasts publish commands from WebSocket connection to all PSHs. But the effective solution should forward *publish* (*channel*, *data*) only to those PSHs where *channel* subscribers are actually present, thus avoiding useless *data* flood on the network. Here comes the most difficult problem of the proposed solution: on every publish WS must know the target PSH subset to forward publish command to.

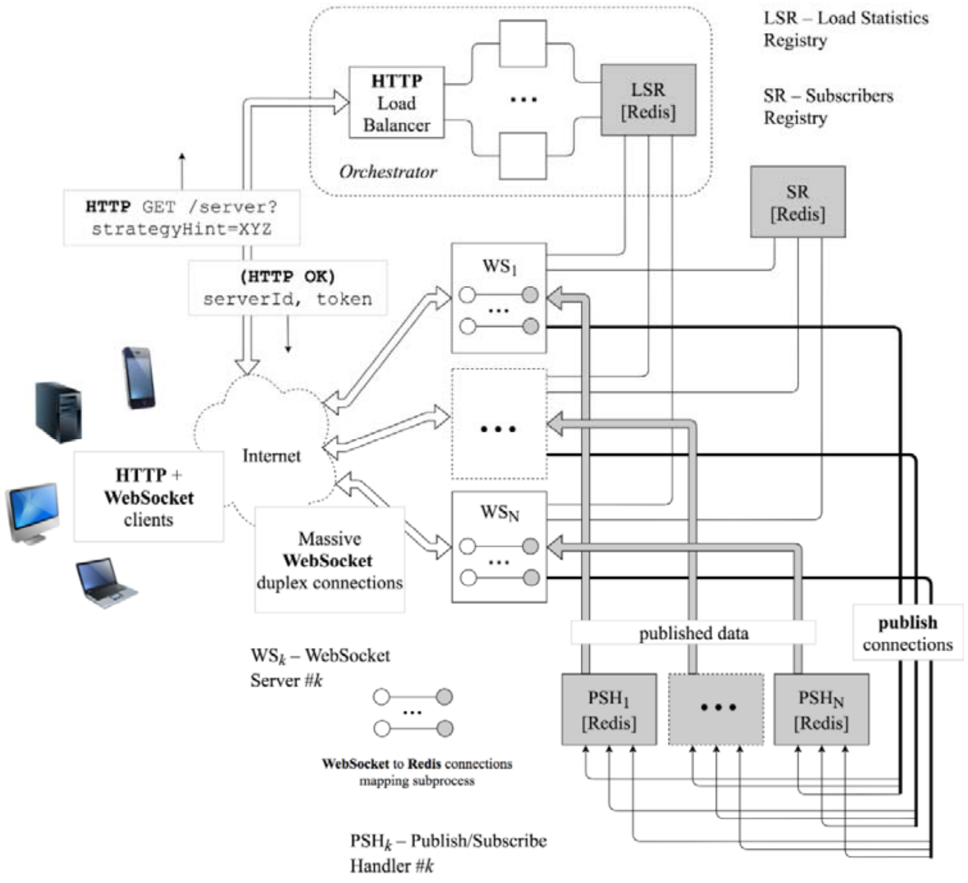
LSR – Load Statistics Registry
SR – Subscribers Registry
HTTP Load Balancer
LSR [Redis]
Orchestrator
SR [Redis]
HTTP GET /server? strategyHint=XYZ
(HTTP OK) serverId, token
WS1
WSN
Internet
HTTP + WebSocket clients
Massive WebSocket duplex connections
published data
publish connections
WSk – WebSocket Server #k
PSH1 [Redis]
PSHN [Redis]
WebSocket to Redis connections mapping subprocess
PSHk – Publish/Subscribe Handler #k

Fig. 4. General scheme for WebSocket scaling with multiple Redis instances.

Therefore, the shared database SR – subscribers' registry – has to be the essential part of the system to provide sets of subscribed PSH instances. We propose the following technique. When any WebSocket connection on WS#k subscribes to *channel*, WS#k creates the special key *Q* on the SR. The name *Q* of this key is concatenation of '#k:' string (where *k* is the actual number) and *channel* string (any other naming scheme will do, if it contains information on both *k* and *channel*). After that WS#k issues Redis increment command [INCRBY Q 1] on the SR. When any WebSocket connection on WS#k unsubscribes from channel, WS#k issues Redis decrement command [INCRBY Q -1] on the SR. If the decrement command returns 0, it means that there is no more *channel* subscribers on this instance (count is zero now), and WS#k must delete the key *Q*: [DEL Q]. To keep the SR registry in actual state WS must update it on every subscribe/unsubscribe command.

When any WebSocket connection on any WS publishes data to *channel*, WS reads *channel* subscription status from the SR. One can use command [MGET keys_list], where *keys_list* is an array of names #1:*channel*, #2:*channel*, etc. The system knows how many servers it uses, so the list can be easily formed by simple iteration. The command will return array of corresponding values (with *nil* for inexistent keys). The publishing WS then iterates this array, selects non-zero entries, and publishes to corresponding PSH instances. Mentioned key lists are typically short (the length depends on *channel* name length, and on total servers count). But published data size can be much larger. That is the reason of using SR: additional round-trip through SR to transfer several kilobytes on every publish is the price of not broadcasting *all* the published data (of unpredictable size) to *all* PSHs.

3. Conclusion and further work

The proposed intelligent WebSocket scaling scheme does not consider the aspects of fault tolerance yet. In addition, we have to construct actual load distribution algorithms for Orchestrator component (the basic approaches were discussed in sections 2.2 – 2.3, but at the present moment they are no more than sketches).

Our scheme does not use any message queuing, it is oriented on real-time instant delivery, but we do not consider it being a drawback for grid computing applications. Fast inter-node temporary data exchange layer (WebSocket-based) is separated from the layer of persistent data storage (DBMS-based). The last is used to take task packets from and to put the results to. Nevertheless, the distributed processing algorithm – the cooperative transformation of every task packet into the result block – needs shared communication bus, allowing every worker node talk to every other node in real-time, due to dynamic and unstable disposition, where persistent message queues with guaranteed delivery would be less relevant solution.

The main goal of the present work was to observe general components of intelligent load-balanced WebSocket-oriented Publish/Subscribe platform. At first stage on this path, we state that WebSocket connections have different behavior types; therefore, an intelligent load balancer has to consider that variance. At second, we propose the negotiation-like mechanism between potential WebSocket client and the load balancing system about the target WebSocket server to set the client on. Client tells its general behaviour features to the system, and the system tries to choose the most appropriate server. The last part covered in our present work relates to the construction principles of scalable multi-server infrastructure with effective inter-server communications. We use Redis instances intensively in our design, because it is a mature system, well tested by the community and praised for its speed and richness of functions.

References

[1] Chorazyk P, Godzik M, Pietak K, Turek W, Kisiel-Dorohinicki M, Byrski A. Lightweight Volunteer Computing Platform using Web Workers. *Procedia Computer Science* 2017;**108**–957.

[2] Chorazyk P. et al. Volunteer computing in a scalable lightweight web-based environment. Computer Assisted Methods in Engineering and Science. *J of Institute of Fundamental Technological Research* 2017;**24**:1-17.

[3] Duda J, Dlubacz W. Distributed evolutionary computing system based on web browsers with JavaScript. In: *Applied Parallel and Scientific Computing*. Springer; 2013. p. 183–191.

[4] WebAssembly Core Specification. Available from: https://www.w3.org/TR/wasm-core-1, last checked 7.08.2018.

[5] Web Workers. W3C Working Draft 24 September 2015. Available from: https://www.w3.org/TR/workers, last checked 7.08.2018.

[6] WebRTC 1.0: Real-time Communication between Browsers. Available from: https://www.w3.org/TR/webrtc, last checked 7.08.2018.

[7] Eugster P, Felber PA, Guerraoui R, Kermarrec AM. The Many Faces of Publish/Subscribe. *ACM Computing Surveys* 2003;**35**:2-114.

[8] Wenjing Fang, Beihong Jin, Biao Zhang, Yuwei Yang, Ziyuan Qin. Design and evaluation of a Pub/Sub service in the cloud. In: *Proceedings of IEEE International Conference on Cloud and Service Computing (CSC)*, 2011; p. 32–39.

[9] RFC 6455 – The WebSocket Protocol. Available from: https://tools.ietf.org/html/rfc6455, last checked 7.08.2018.

[10] The WebSocket API. W3C Candidate Recommendation 20 September 2012. Available from: https://www.w3.org/TR/websockets, last checked 7.08.2018.

[11] NGINX WebSocket Performance Tests. Available from: https://www.nginx.com/blog/nginx-websockets-performance, last checked 7.08.2018.

[12] Using NGINX as a WebSocket Proxy. Available from: https://www.nginx.com/blog/websocket-nginx, last checked 7.08.2018.

[13] Redis. Available from: https://redis.io, last checked 7.08.2018.

[14] Redis Cluster Specification. Available from: https://redis.io/topics/cluster-spec, last checked 7.08.2018.