Towards using SCTP as a Data Transport Protocol for Data-Intensive Batch Jobs

Lucian Ghinea, Mugurel Ionut Andreica, Vlad Olaru, Nicolae Tapus

Towards using SCTP as a Data Transport Protocol for Data-Intensive Batch Jobs

Lucian Ghinea, Mugurel Ionuţ Andreica, Vlad Olaru, Nicolae Ţăpuş
Computer Science Department
Politehnica University of Bucharest
Bucharest, Romania
ghinea.lucian@gmail.com, mugurel.andreica@cs.pub.ro, vladolaru@gmail.com, nicolae.tapus@cs.pub.ro

***Abstract*—In this paper we investigate the usage of the Stream Control Transmission Protocol (SCTP) as a data transport protocol for client-initiated data-intensive computations. The considered environment consists of job schedulers, storage nodes, computational nodes (workers) and clients. The clients submit jobs to the schedulers, which split them into multiple computation units and schedule these computations on the available workers. The data which needs to be processed by the computation units will be transferred from the storage nodes to the computational nodes using SCTP. Experimental evaluations considered images as the data to be processed and simple image processing operations as computations.**

Keywords-data-intensive computations; SCTP; SOAP; workers; image processing

I. INTRODUCTION

In this paper we investigate the usage of the Stream Control Transmission Protocol (SCTP) [9] as a data transport protocol for data-intensive computations. The entities which are part of our considered environment are the following:

- job schedulers
- computational nodes (workers)
- storage nodes
- clients

The job schedulers receive job submissions from clients. A job is submitted from a client to a single job scheduler, selected among the available ones, and consists of performing a specific operation on some specific input data. The input data is located on the storage nodes and, thus, it is referenced by an identifier within the client's submission. The storage nodes are, essentially, organized in a distributed storage system which may present varying degrees of complexity.

The operation which needs to be performed may be split by the scheduler into multiple independent computation units, which may be executed in parallel. Each such computation unit will be executed on a separate part of the input data (although our system does not exclude the possibility of overlapping input data for multiple computation units). Although we will only consider the case in which the computation units may be executed independently in this paper, our system allows for the job to be split into a set of computation units where some units are dependent upon other units (i.e. the job is split into a directed acyclic graph of computation units, where the output of some units is part of the input of other units).

The scheduler is responsible for scheduling the execution of computation units on the available workers and for scheduling the data transfer of the corresponding input data from a storage node which has the data to the worker which requires that data and for scheduling the data transfer of the output data of each computation unit from the worker where it was running to one or more storage nodes (selected by the scheduler).

The computation units may not be split any further. A computation unit needs to be executed sequentially, by a single worker.

Besides actually transferring the data files which need to be processed, our system's entities need to communicate with each other. The clients need to submit their jobs, while the job schedulers, storage nodes and workers need to cooperate. This communication is not data-intensive, so we chose to use web services for this part (based on SOAP).

The rest of this paper is structured as follows. In Section II we discuss related work. In Section III we present the exact architectural details of the system which we considered for testing the impact of SCTP as a data transport protocol. In Section IV we present experimental results. In Section V we conclude and discuss future work.

II. RELATED WORK

A. Volunteer Computing Systems

Volunteer computing systems are distributed systems in which individuals donate their computing resources (e.g. CPU cycles and storage space) to one or more projects. The first volunteer computing system was "Great Internet Mersenne Prime Search", started in January 1996, followed in 1997 by distributed.net. In 1998 a series of academic Java-based projects were started, like Bayanihan, Popcorn, Superweb and Charlotte [3-6].

In 1999 SETI@home [7] and Folding@home [8] were launched. These projects became very well known and attracted hundreds of thousands of users.

Between 1998 and 2002 a series of commercial projects were launched, whose purpose was to develop solutions based on sharing the resources of multiple devices (e.g. Popular Power, Porivo, Entropia and United Devices).

Most of these systems have the same base structure: a client program runs on a "volunteer" system. Periodically, the client contacts the project's server, requesting tasks and sending back results. Volunteer computing systems need to find solutions to several aspects regarding their functionality: heterogeneity, variable number of clients, variable availability of the clients, etc.

B. *XtremWeb*

XtremWeb [2] is a global computing architecture which aims to take advantage of multiple types of workers, with various performance levels. Although it is not particularly focused on data-intensive computing, its general architecture is similar in structure and scope with that of our system.

C. *SOAPExpress*

SOAPExpress [1] is a SOAP web services engine in which transport of the data is performed over SCTP instead of TCP. Performance improvements of up to 56% were noticed. The architecture of SOAPExpress is presented in Fig. 1.

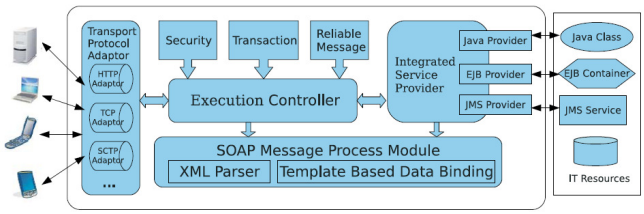


Figure 1. SOAPExpress architecture [1].

D. *Stream Control Transmission Protocol*

SCTP is a relatively new data transport protocol which provides features like: multi-streaming, multi-homing, reliable delivery of packets, preservation of message boundaries, congestion control and many others. It is a connection-oriented protocol, like TCP, but within a connection (called SCTP association) there are multiple independent streams. SCTP has the potential of behaving better than TCP at least on the following two accounts:

- A SCTP association with *N* streams consumes fewer resources than *N* parallel TCP connections.
- Sending packets over multiple streams avoids head-of-line blocking and may increase throughput (if a packet on some stream is lost, the congestion window is reduced only for that stream, while the other streams are unaffected; moreover, a lost packet only delays further other packets sent on the same stream).

Many projects in which SCTP is used as a data transport protocol instead of TCP or other data transport protocols were discussed in the scientific literature. The usage of SCTP as a data transport protocol for web servers was considered in [10, 11]. Experiments of SCTP vs. TCP for high-speed intra-cluster communication (with applications to cluster-based data acquisition systems in mind) were performed and presented in [12]. SCTP was also compared against TCP for communication between MPI-based processes [13].

III. ARCHITECTURAL DETAILS OF OUR SYSTEM

In this section we will describe in more detail the functions of and the interactions between our system's entities.

A. *Job Schedulers and Storage Nodes*

Because dynamic coordination of the data transfers is difficult when the scheduler and the storage nodes are separate, we decided to combine the job scheduling and storage functions together, on the same machine. Thus, our system will consist only of *"servers"* (a server is both a job scheduler and a storage node), workers (computational nodes) and clients. Fig. 2 presents the generic architecture of our system, with only one server depicted.

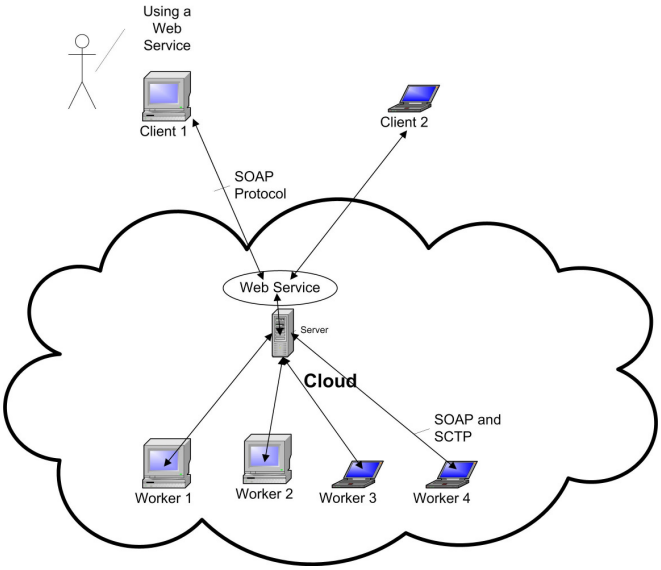


Figure 2. Generic system architecture.

The server stores both data (files) and the code of the operations which can be performed on the data. The server interacts with clients and workers. The client-server interactions consist of the following steps:

- The server receives a job submission from a client.
- The server splits the job into multiple computation units and schedules them for execution on the available workers.
- When all the computation units are finished, the server assembles the final result from the results of each computation unit and sends the answer to the client (note that the answer does not necessarily contain the final result itself; instead, it may contain a data identifier with which the

client may later retrieve the result). Alternatively, the server may not send an answer to the client (e.g. because the client may not be accessible from the server); instead, the client is responsible for polling the server regarding the status of its submission (when the job is completed, the client will get the answer as the result of the polling).

All the communication between the server and the clients is performed by using web services (based on SOAP).

The server maintains a list of available workers, together with information regarding their performance level and other parameters. The way the server splits a job into computation units and schedules these units on the available workers is part of the server's *job scheduling strategy*. From our perspective, we will be interested in the following two aspects of this strategy:

- Static or dynamic splitting of the job into computation units (e.g. all the units are generated at the beginning, or they are generated dynamically, according to the results and monitoring data concerning already generated units)
- Static or dynamic association of computation units to workers: the number of computation units sent to each worker may be decided in the beginning (statically), or may be adjusted dynamically, according to the results of the completed computation units and according to monitoring data

The worker-server interaction consists of the following steps:

- When the worker starts, it registers itself to one or more (or maybe even all the) job schedulers. During the registration phase, the worker sends to the job scheduler information about how to be contacted in order to process jobs (e.g. its own web service address, SCTP IP address and port, etc.), as well as performance information (e.g. RAM size, hard-disk size, number of processors/cores, processor frequency, etc.), as well as the maximum number of processing threads it is willing to assign for running computation units. Moreover, the worker may download the code for some or all of the operations that may be performed on the data from the server. It is possible for a worker to be able to execute only a subset of the total number of operations which the clients may request. A worker will only be considered for scheduling a computation unit if it is capable of executing the corresponding operation.
- The worker will announce each scheduler to which it registered whenever there is a change in its performance characteristics, or, optionally, a job scheduler may ask for updated performance characteristics from each worker at any time.
- After a job is received from a client and a worker is selected for processing the job, the server will establish an SCTP association to each selected worker. The number of SCTP streams used (both incoming and outgoing) will be at most equal to the number of (still) available processing threads on that worker. Once the association is open, the worker reserves a processing thread for each SCTP stream and, thus, those threads will not be available anymore until the association is closed.
- The server will send computation units on SCTP streams. A computation unit will be self-describing. It will properly identify the operation which needs to be performed and it will also contain the data on which the operation needs to be performed. A computation unit is sent on a single SCTP stream, as a sequence of one or more consecutive packets.
- The worker will send back the result of each computation unit to the server, on one of the SCTP streams (possibly the same one on which the unit was received). Each computation unit should have a unique identifier assigned by the server and the worker's result will contain the identifier of the corresponding computation unit.

The sending of computation units and the receiving of their results is performed by using SCTP. All other communication between a server and a worker is performed by using SOAP-based web services.

B. *Workers (Computational Nodes)*

A computational node (worker) executes computation units. It interacts directly only with the servers and not with the clients. When a worker starts, it registers to some of the existing servers (how a worker finds out a list of servers is outside the scope of this paper). Only those servers to which the worker registered will be able to send computation units to it. The interaction between a worker and a server was described in the previous subsection. Here we will discuss the communication and computation unit processing aspects.

Each worker has a thread which listens for incoming SCTP connections. Once an SCTP association is established, it is handled to a thread from a pool of *packet receiving threads*. Each thread from the pool is responsible for receiving packets from all the streams of a subset of SCTP associations which are associated to it (by using the socket *Selector* paradigm).

The worker also has a pool of available processing threads. When a SCTP association with M streams is started, M processing threads are removed from the pool of processing threads and a sub-pool containing these threads is created. Then, whenever a computation unit is received from a stream of the SCTP association, the unit is placed in a dedicated queue. The M threads from the sub-pool take the units from the queue and execute them. When the execution of a computation unit is finished, the thread which executed the unit sends the result back to the server on the SCTP association (using the same stream on which the unit was received). Since multiple

threads may try to send data on the same SCTP stream simultaneously, a synchronization mechanism for each stream of an association is used. Alternatively, each of the *M* threads could have been statically assigned to one of the *M* streams of the SCTP association. Each thread would process only computation units received on the assigned stream and would send the results back on the same stream. No per-stream-synchronization mechanism would be required in this case. However, we did not implement this option. When the association is closed, the *M* threads are returned to the pool of available processing threads. See Fig. 3 for a graphical description of these steps.

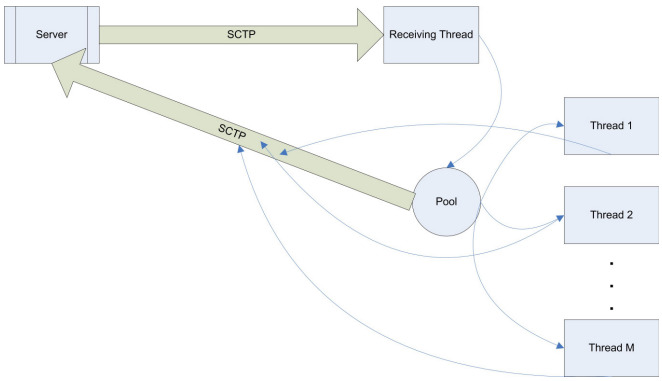


Figure 3. Processing of computation units by a worker.

C. Client

The client selects a job scheduler and submits a job to it (how the client finds out about or selects a job scheduler or what format the job submission has is outside the scope of this paper). The client may receive a response from the scheduler in a synchronous or asynchronous manner. In the synchronous manner, the client blocks until the response arrives. The blocking duration may be very long, depending on the complexity of the job and on the size of the processed data. SOAP allows for a web service client to receive asynchronous responses, so this is the option we selected.

IV. Experimental Results

A good example of data-intensive computations consists of image processing operations. Thus, in our system, a job submitted by a client requests a specific operation to be applied on an image (stored on the storage nodes). An image processing operation is not necessarily an atomic operation. In fact, most image processing operations can be split into multiple independent computation units (tasks) which are performed on separate parts of the image. The parallelization may be as finely grained as we want. Thus, there is a large potential for parallelizing a job and also for parallelizing the data transfer of an image from the storage nodes to the workers performing the computation.

The purpose of our experiments was to analyze the impact of using multi-stream SCTP on a set of simple image processing tasks. We tested the cases when the number of tasks sent to each worker is the same, or is different depending on the number of threads available on each worker. Each task (computation unit) consisted of applying a *blur* operation on a 19 KB GIF file. The size overhead regarding the description of the operation to be performed is negligible compared to the size of the data being processed. We always used the same file for each task. The client request mentioned the name of the file and the number of times the operation should be performed (thus effectively deciding the number of tasks).

We used 4 physical machines, each of them running the Ubuntu operating system within a virtual machine:

- S1 – Windows 7 64 bits, Core 2 Duo 2.53 Ghz , 4GB RAM DDR 3, VM Ubuntu (2GB RAM, 2 processors, HDD 10GB)
- C1 – Windows XP 32 bits, Core 2 Duo 2.53 Ghz, 4GB RAM DDR3, VM Ubuntu (1.5 GB RAM, 2 processors, HDD 10 GB)
- C2 – Windows XP 32 bits, Core 2 Duo 2.53 Ghz, 4GB RAM DDR3, VM Ubuntu (1.5 GB RAM, 2 processors , HDD 10 GB)
- C3 – Windows XP 32 bits, Pentium 4, 3Ghz, 4GB RAM DDR3, VM Ubuntu (1.5 GB RAM, 1 processor, HDD 10 GB)

In all the tests, S1 was used for running the server (the web service and communication and management module, plus storing the image files and the code of the image processing operations). We implemented the server program in Java, using OpenJDK 7 [14] (which is the first JDK version with support for SCTP) and the Apache web server [16] plus Apache Axis [15] for the web service. We did not run any tests with multiple servers (e.g. where a worker is registered at both servers and receives tasks from both servers in parallel).

C1, C2 and C3 were used as machines for running independent workers. The worker programs were also implemented in Java, using the same technologies as for the server (OpenJDK 7 and the Apache web server plus Apache Axis for the web service). The number of available threads could easily be configured from a configuration file, as well as the ports on which SCTP communication takes place.

C1 and C2 had an identical hardware structure and ran the same virtual machine. C3, however, had a lower performance. By changing the data sent by the workers during the registration phase we could analyze the impact of improper worker choices.

All the measured times are expressed in milliseconds (ms) and were computed as an average over multiple instances of the same test (between 5 and 10). Time measurement always started as soon as the server received the job from the client and ended as soon as it received all the processed images back.

The number of SCTP streams used in order to communicate between a server and a worker was always chosen to be equal to the number of processing threads advertised as available by the worker (thus, all the available processing threads of a worker were fully utilized by the test job).

A. *Reference Test*

We considered that there was only one worker, and SCTP communication took place over only one stream (similarly to TCP). C1 was used as the worker. Results are presented in Fig. 4.

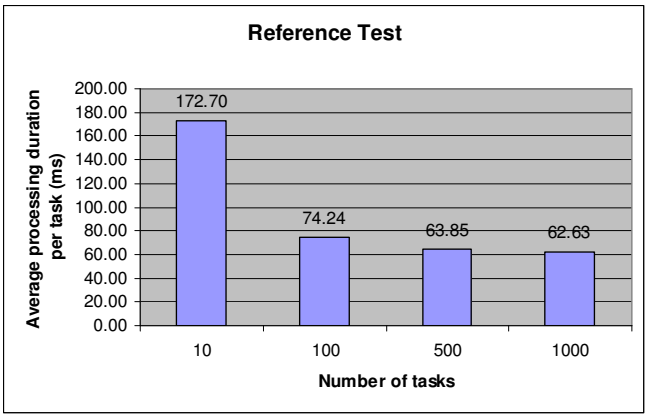
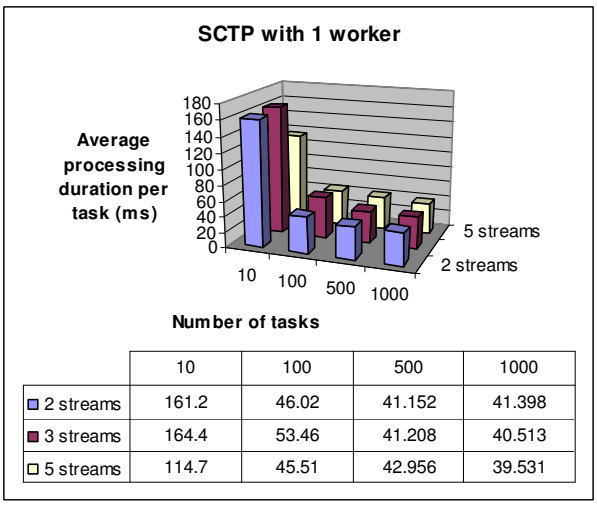


Figure 4. Average processing duration per task for one worker (C1) and one SCTP stream.

We notice a decrease of the average processing duration per task once the total number of tasks increases. This decrease is a reflection of the fact that the influence of the SCTP association establishment overhead is reduced over multiple tasks.

B. *SCTP with 1 Worker*

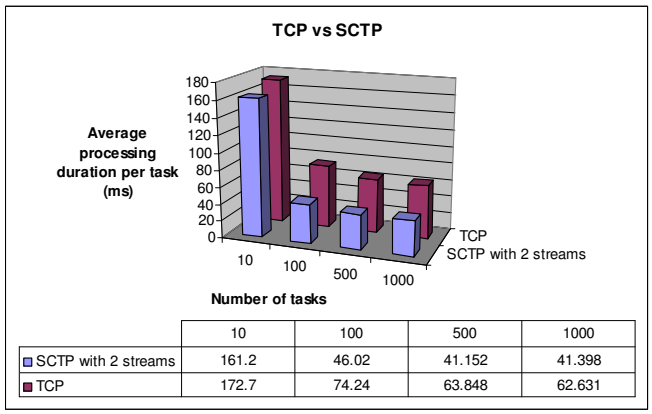
We considered only one worker, for which we varied the number of tasks and the number of available threads. We used C1 for running the worker. We also considered the case when the worker advertises more threads than the number of available processors. The number of SCTP streams is always equal to the number of available processing threads.



	10	100	500	1000
2 streams	161.2	46.02	41.152	41.398
3 streams	164.4	53.46	41.208	40.513
5 streams	114.7	45.51	42.956	39.531

Figure 5. Average processing duration per task for one worker (C1) and variable number of SCTP streams.

We notice (Fig. 5) that by varying the number of SCTP streams (2, 3 and 5 streams) the average processing duration per task decreases significantly.



	10	100	500	1000
SCTP with 2 streams	161.2	46.02	41.152	41.398
TCP	172.7	74.24	63.848	62.631

Figure 6. TCP vs SCTP with 2 streams – Average processing duration per task.

In Fig. 6 we compare TCP and SCTP with two streams. We notice that SCTP outperforms TCP as the number of tasks increases (with over 30% for more than 100 tasks – see Fig. 7).

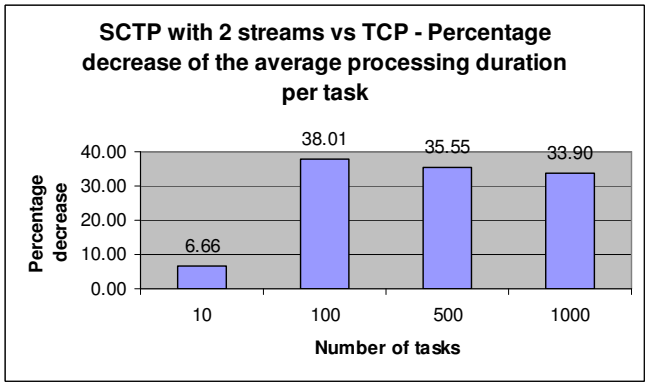


Figure 7. TCP vs SCTP with 2 streams – Percentage decrease of the average processing duration per task.

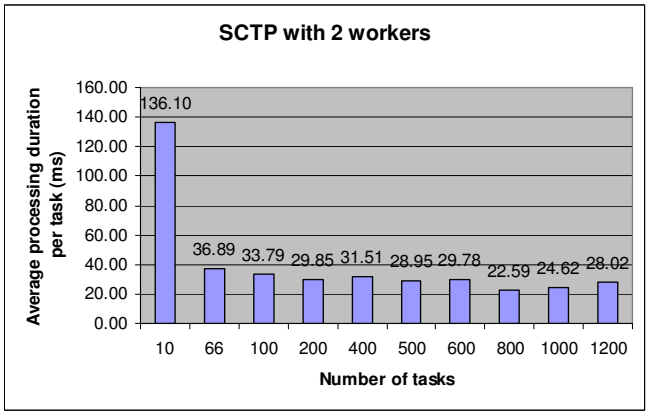


Figure 8. Average processing duration per task for SCTP with 2 streams per worker and 2 workers.

The same test was run with C2 as the worker, in order to compare the results of two virtually identical machines. The results obtained for C2 varied by at most 5% from the results obtained for C1, which is considered acceptable.

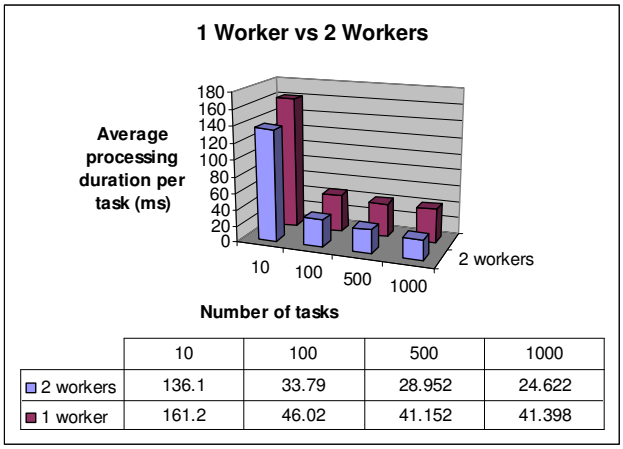
C. SCTP with 2 Workers

We considered SCTP with 2 streams for this case, because each machine only has two processors and there are also other threads running (e.g. the web service thread, packet receiving threads, etc.).

For this test, each of the two workers (C1 and C2) received half of the total number of tasks, considering that the previous test showed that the performances of C1 and C2 are very similar. We notice improved results (see Fig. 8) compared to the one worker – two streams case tested earlier.

D. 1 Worker vs 2 Workers

Considering the previous results, we wanted to analyze the performance improvement when using two workers (C1 and C2) instead of one (C1). All workers used two SCTP streams.



	10	100	500	1000
2 workers	136.1	33.79	28.952	24.622
1 worker	161.2	46.02	41.152	41.398

Figure 9. Average processing duration per task for SCTP with 2 streams per worker. 1 worker vs 2 workers.

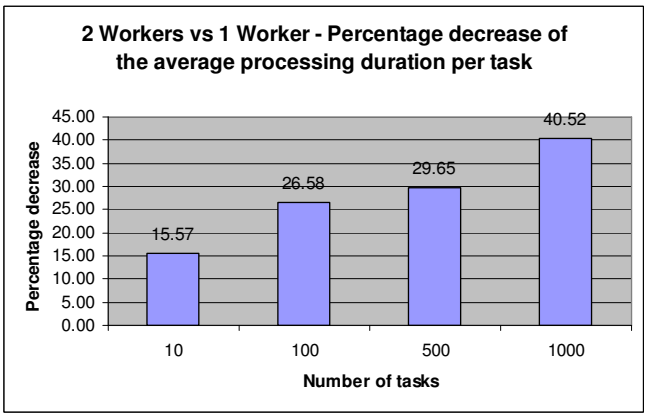


Figure 10. SCTP with 2 streams per worker, 2 workers vs 1 worker – Percentage decrease of the average processing duration per task.

Using two workers is faster than using just one (see Fig. 9 and Fig. 10), but the average processing duration per task does not drop all the way to 50%. Instead, at best, a 40% decrease of running time is noticed.

We also tested 2 SCTP workers (with 2 streams each) against one TCP worker. The percentage decrease of the average processing duration per task for the SCTP case can be seen in Fig. 11.

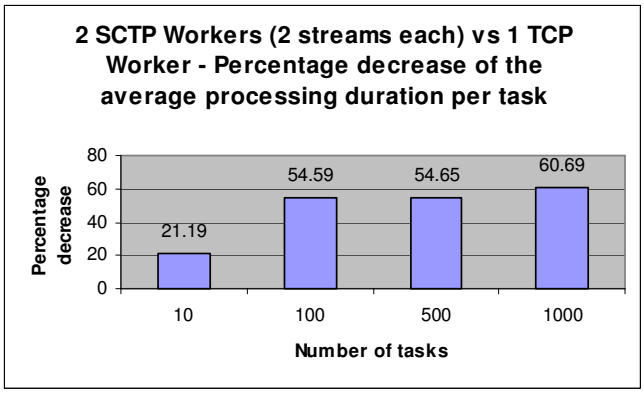
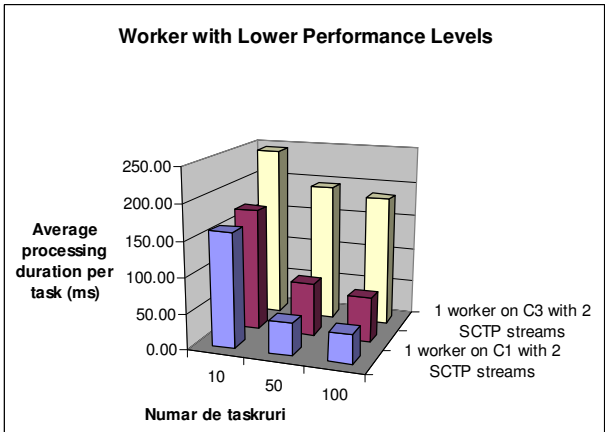


Figure 11. SCTP with 2 streams per worker and 2 workers vs 1 TCP worker – Percentage decrease of the average processing duration per task.

E. Using a Worker with Lower Performance Levels

In this test we used the C3 machine in order to run a worker. We notice a notable performance difference between a worker running on C1 and one running on C3 (see Fig. 12).



	10	50	100
1 worker on C1 with 2 SCTP streams	161.20	46.02	41.15
1 worker on C1 with 1 SCTP stream	172.70	74.24	63.85
1 worker on C3 with 2 SCTP streams	244.30	195.28	184.61

Figure 12. Using a worker with lower performance levels. Average processing duration per task.

F. *Using an Extra Worker with Lower Performance Levels together with Two Higher Performance Workers*

Although C3 has lower performance compared to C1 and C2, we still want to use it as part of the system, together with C1 and C2.

1. *Equal division of the number of tasks*

We used C1, C2 and C3 for running workers, with two threads (and two SCTP streams) each.

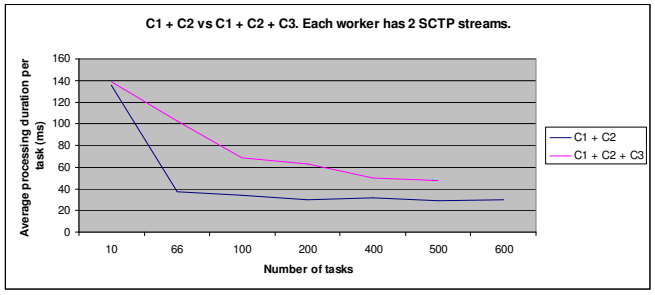


Figure 13. 2 high performance workers vs 2 high performance workers plus a low performance worker – Average processing duration per task when each worker receives the same fraction of the number of tasks.

We notice that by introducing C3 without considering its performance the overall performance of the system decreases (see Fig. 13).

2. *Proportional division of the number of tasks*

In this test each worker received a number of tasks which was proportional to the number of available threads. We considered C1 and C2 as having two threads each and C3 as having only one thread. We notice that although C3 only received half of the tasks each of C1 and C2 received, we still do not obtain a performance improvement (see Fig. 14).

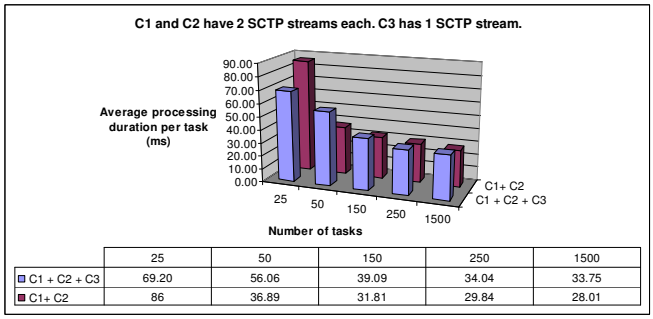
We considered next 3 processing threads for each of C1 and C2 and only one thread for C3. In this case we notice that C3 brings a performance improvement (Fig. 15). Basically, any extra worker may increase the system performance, as long as its share of the total number of tasks is appropriately selected according to the worker's performance.

From the server's perspective, allocating an appropriate number of tasks to each worker is a very important step. Perhaps a better approach would be for the server to establish a pipeline of tasks. Initially, only a fraction of the tasks is statically allocated among the workers. Then, as soon as a task is finished by a worker, a new one is sent to the same worker, thus obtaining a dynamic adaptation.

V. Conclusions and Future Work

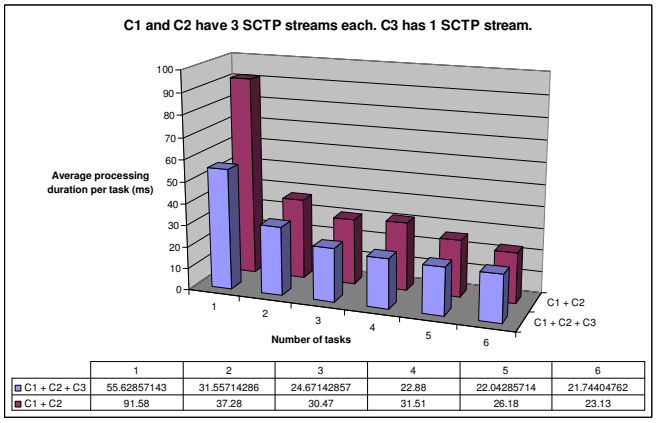
In this paper we investigated the usage of SCTP as a data transport protocol for data-intensive parallelizable computations. The experimental results showed that SCTP may provide much better performance levels than TCP (or other TCP-based protocols), but using it needs to be in accordance with the performance levels of the computational nodes (workers). Lack of carefulness in scheduling the computation units (tasks) over the available workers may lead to the nearly paradoxical case in which having extra workers actually hurts the performance.

As future work we intend to consider multiple job scheduling policies, both static and dynamically adjustable, and see how they can be best combined with the usage of SCTP as the data transport protocol. Moreover, we also intend to separate the job scheduling and storage functions on separate sets of machines. In this case, coordinated scheduling of data transfers from and to the storage nodes may also be relevant in order to achieve optimal performance.



	25	50	150	250	1500
C1 + C2 + C3	69.20	56.06	39.09	34.04	33.75
C1+ C2	86	36.89	31.81	29.84	28.01

Figure 14. 2 high performance workers vs 2 high performance workers plus a low performance worker – Average processing duration per task when each worker receives a fraction of the number of tasks which is proportional to the number of available processing threads (or SCTP streams). C1 and C2 have 2 threads each and C3 has only one thread.



	1	2	3	4	5	6
C1 + C2 + C3	55.62857143	31.55714286	24.67142857	22.88	22.04285714	21.74404762
C1 + C2	91.58	37.28	30.47	31.51	26.18	23.13

Figure 15. 2 high performance workers vs 2 high performance workers plus a low performance worker – Average processing duration per task when each worker receives a fraction of the number of tasks which is proportional to the number of available processing threads (or SCTP streams). C1 and C2 have 3 threads each and C3 has only one thread.

Acknowledgement

The work presented in this paper has been partially funded by CNCS-UEFISCDI under research grants ID_1679/2008 (contract no. 736/2009), PN II – IDEI program, and PD_240/2010 (contract no. 33/28.07.2010), PN II – RU program, and by the Sectoral Operational Programme Human Resources Development 2007-2013 of the Romanian Ministry of Labour, Family and Social Protection through the Financial Agreement POSDRU/89/1.5/S/62557.

REFERENCES

[1] N. Wang, M. Welzl, and L. Zhang, "A High Performance SOAP Engine for Grid Computing", Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, vol. 2, pp. 1-8, 2009.

[2] F. Cappello, et al., "Computing on Large-scale Dstributed Sstems: XtremWeb Architecture, Programming Models, Security, Tests and Convergence with Grid", Future Generation Computer Systems, vol. 21, pp. 417-437, 2005.

[3] L. F. G. Sarmenta, "Bayanihan: Web-Based Volunteer Computing Using Java", Lecture Notes in Computer Science, vol. 1368, pp. 444-461, 1998.

[4] O. Regev and N. Nisan, "The POPCORN Market - an Online Market for Computational Resources", Proceedings of the First International Conference on Information and Computation Economies, pp. 148-157, 1998.

[5] A. D. Alexandrov, M. Ibel, K. E. Schauser, and K. E. Scheiman, "SuperWeb: Research Issues in Java-Based Global Computing", Concurrency: Practice and Experience, vol. 9 (6), pp. 535-553, 1997.

[6] A. Baratloo, M. Karaul, Z. Kedem, and P. Wyckoff, "Charlotte: Metacomputing on the Web", Future Generation Computer Systems, vol. 15 (5-6), 1999.

[7] Folding@home project. http://folding.stanford.edu/English/Main

[8] Seti@home project. http://setiathome.ssl.berkeley.edu

[9] R. Stewart, et al., "RFC2960 - Stream Control Transmission Protocol", 2000. Available online at http://tools.ietf.org/html/rfc2960.

[10] V. Olaru, M. I. Andreica, and N. Ţăpuş, "Using the Stream Control Transmission Protocol and Multi-Core Processors to Improve the Performance of Web Servers", Proceedings of the 13th IEEE International Conference on High Performance Computing and Communications (HPCC), pp. 135-144, 2011.

[11] P. Natarajan, J. R. Iyengar, P. D. Amer, and R. Stewart, "SCTP: An Innovative Transport Layer Protocol for the Web", Proceedings of the 15th International Conference on World Wide Web, pp. 615-624, 2006.

[12] M. Kozlovszky, T. Berceli, and L. Kutor, "Analysis of SCTP and TCP based Communication in High-speed Clusters", Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment, vol. 559 (1), pp. 85-89, 2006.

[13] H. Kamal, B. Penoff, and A. Wagner, "SCTP versus TCP for MPI", Proceedings of the 2005 ACM/IEEE Conference on Supercomputing, pp. 30-43, 2005.

[14] OpenJDK 7. http://openjdk.java.net/projects/jdk7/

[15] J. Goodwill, "Apache Axis Live: A Web Services Tutorial", SourceBeat, LLC, 2004.

[16] B. Laurie and P. Laurie, "Apache: The Definitive Guide", O'Reilly, 2003.