Merging the Publish-Subscribe Pattern with the Shared Memory Paradigm

Loïc Cudennec

Merging the Publish-Subscribe Pattern with the Shared Memory Paradigm

Loïc Cudennec
[0000−0002−6476−4574]

CEA, LIST,
F-91191, PC 172, Gif-sur-Yvette, France
loic.cudennec@cea.fr

Abstract. Heterogeneous distributed architectures require high-level abstractions to ease the programmability and efficiently manage resources. Both the publish-subscribe and the shared memory models offer such abstraction. However they are intended to be used in different application contexts. In this paper we propose to merge these two models into a new one. It benefits from the rigorous cache coherence management of the shared memory and the ability to cope with dynamic large-scale environment of the publish-subscribe model. The publish-subscribe mechanisms have been implemented within a distributed shared memory system and tested using an heterogeneous micro-server.

Keywords: S-DSM · Publish-Subscribe · Heterogeneous Computing.

1 Introduction

Distributed heterogeneous architectures are considered as a solution for different computing contexts that provides computational performance while saving the energy consumption. A mix of high-performance and low-power computing nodes are used in HPC and data centers, and a mix of low-power processors, specific accelerators (eg for deep learning), GPUs and FPGAs will be used in future embedded devices for autonomous vehicles or future industry. As for current distributed heterogeneous architectures, a part of the main challenges is the efficient programmability.

In such architectures, memories are physically distributed among the nodes, which makes the management of data between tasks more complex. For example it does not allow regular parallel applications with direct access to shared data: inter-node access requires explicit message passing from the user, which is usually addressed using hybrid programming. Distributed shared memory systems (DSM) offer an abstraction layer by federating memories into a global logical space. Data management is hidden by the runtime. The shared memory paradigm is convenient for programming HPC applications with quite a static topology and regular access patterns. However it is not well adapted to event-based applications in which volatile tasks get notified whenever an object state changes.

Event-based applications are popular in web services, wireless networks and peer-to-peer (P2P) systems. It can be used to monitor a sensor, the output of a computation or a decision system. This makes the programming paradigm well adapted to new fields of computation such as the industry and the automotive world. The publish-subscribe mechanism relies on a set of mutable objects that can publish notifications to a set of subscribers. This paradigm fits to dynamic applications with unexpected and volatile access to shared data. However, what makes the strength of the paradigm is also a limitation it terms of data coherence management. First, data sharing is usually immutable, meaning that data is modified by the publisher while the subscribers are read-only. Second, the protocol is loosely coupled and the data coherence model is very permissive. It is quite difficult to ensure that the published version read by the subscribers is the current version in the distributed system, according to the causal model.

Upcoming distributed heterogeneous platforms will also run heterogeneous applications in terms of programming models. For example we can mix HPC simulation code with event-based monitoring GUI. In this paper, we propose to merge the publish-subscribe model with the shared memory model. We start from an in-house DSM, we extend the API and implement some distributed mechanisms on the atomic piece of shared data to raise publishing events. We implement a video processing application with the regular shared memory paradigm and the proposed mixed paradigm. Both implementations are then evaluated on a Heterogeneous Christmann RECS|Box Antares Microserver.

2 Shared Memory, Synchronization Objects and Events

Shared memory is a convenient programming paradigm in which a set of tasks can transparently access a set of data. The implementation of such system is quite straightforward on a physically shared memory but reveals to be complex on a distributed architecture. Software-Distributed Shared Memory (S-DSM) is used to federate distributed physical memory and provide a high level of abstraction to the application. In a S-DSM, the system is in charge of the location, the transfer and the management of multiple copies of data. It also provides objects and primitives to synchronize task execution and concurrent accesses. DSM in general have been studied since the late eighties to federate computer memories [12,7,8,14], clusters [1,3,16,17,18] and grids [4].

In a previous work [9], a S-DSM is proposed for heterogeneous distributed architectures such as micro-servers. This system allows tasks to allocate and access memory in a shared logical space. This is a super-peer distributed topology in which the user code is executed by S-DSM clients, connected to S-DSM servers. Allocated data can be split into *chunks* of any size, the atomic piece of data managed by the S-DSM. While the data is always locally allocated in a contiguous memory space on the clients to allow pointer arithmetic, the corresponding chunks are not necessarily contiguous in the shared logical space and can also be managed independently by the S-DSM servers afterwards. Accessing shared data follows the entry consistency scope paradigm [7]. In this paradigm, access

must be protected in the user code using 1) the *READ* or *WRITE* primitive to enter the scope and 2) the *RELEASE* primitive to exit the scope. Outside this scope, data consistency is not guaranteed. The S-DSM can deploy different data coherence protocols on different chunks. In this paper we use the home-based 4-state MESI protocol [10], which allows a single writer and multiple readers (SWMR). The chunk metadata management is distributed among the S-DSM servers by calculating a modulo on the chunk id.

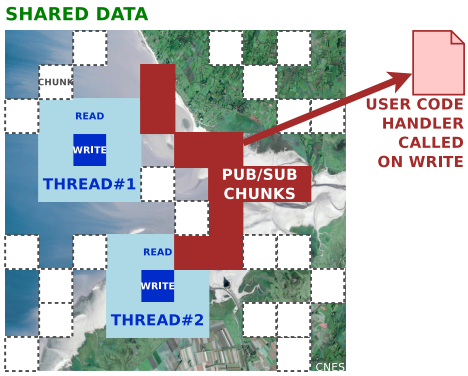


Fig. 1: *Event coding in shared memory. Application to the parallel processing of tiles.*

As a motivating example, we consider a parallel HPC application as illustrated in Figure 1. The purpose is to calculate the sea level for each time step by applying a wave propagation model. After each iteration, some specific places on the map are monitored to detect if there is a threat to the population. The base map is represented as a set of chunks in the shared memory, each chunk covering a square surface (a tile). Several threads navigate the chunks to update values. Some other threads monitor the critical chunks (represented in red color). One realistic constraint is that it is not possible to modify the HPC code to manage the critical aspect of the calculation. Instead, we expect a smooth and non-intrusive integration of the critical code regarding the HPC code.

A first approach is to use one rendez-vous and one monitoring thread per critical chunk. Each time a HPC thread calculates a critical chunk, it invokes the corresponding rendez-vous and releases the critical threads. This is a static approach regarding the number of critical chunks and this requires to modify the HPC code to invoke rendez-vous. A second approach is based on polling the critical chunks: a set of monitoring threads continuously access the critical chunks for new values. It possibly generates useless requests and network activity in the S-DSM. It is also prone to skipping some updates between two accesses, unless implementing a dual rendez-vous producer-consumer pattern.

A more elegant approach is to rely on a event-based publish-subscribe (PS) mechanism. Monitoring threads subscribe to critical chunks and get notified each time the chunk has been modified. This approach is transparent for the HPC code and allows dynamic subscription of threads to critical chunks. In this paper, we propose to design and implement this publish-subscribe mechanism within the S-DSM runtime.

3 Event Programming with Memory Chunks

The publish-subscribe paradigm is defined by a set of mutable objects (publishers) and a set of subscribers. There is a many-to-many relationship between publishers and subscribers. Each time the mutable object is changed, it publishes the information to all its subscribers. The information can be a simple notification, an update or the complete data. We propose to merge the publish-subscribe model with the shared memory programming model, with a few modifications to the user API and the S-DSM runtime. The basic idea is to use chunks as mutable publishing objects and to extend the distributed metadata management for chunk coherence on the S-DSM servers with publish-subscribe metadata management. We consider the three following listings.

```
1    void main_publisher() {
2      mychunk = MALLOC(chunkid, size); /* allocate shared data in S-DSM */
3      WRITE(mychunk);                  /* ask for the write lock */
4       foo(mychunk);                   /* in this scope it is possible to write chunk */
5      RELEASE(mychunk);                /* release the write lock */
6    }
```

```
1    void main_subscriber() {
2      mychunk = LOOKUP(chunkid);        /* fetch information about previously allocated chunk */
3      SUBSCRIBE(mychunk, subscriber_handler, parameters);
4                                        /* subscribe to the chunk with given user handler */
5    }
```

```
1    void subscriber_handler(chunk, parameters) {
2      WRITE(chunk);                     /* ask for the write lock */
3       foo(chunk, parameters);          /* in this scope it is possible to read and write chunk */
4      RELEASE(chunk);                   /* release the write lock */
5      UNSUBSCRIBE(chunk);               /* unsubscribe to the chunk, this handler wont be call */
6                                        /* afterwards, all publish notifications are discarded, */
7                                        /* including the RELEASE in this function */
8    }
```

The first listing implements the publisher role. This code only makes use of regular S-DSM primitives. The publish-subscribe API is used by subscribers, as presented by the second listing. The subscribe primitive registers a user handler (a pointer to a local function) and some user parameters to a given chunk. Each time the chunk is modified -from anywhere in the S-DSM- this handler is called on the subscribing task. Finally, a handler function example is given in the third listing. Within the function it is possible to access shared data, subscribe to other chunks and unsubscribe to any chunk. The same handler function can be used to subscribe different chunks.

Publish-subscribe events are *sequentialized* on each client and the corresponding handlers are called in the notification message delivering order. This choice has several ins and outs. First, the user code is easier to write because there is no local concurrency to manage. Second, this implies a tight design of the message handling in the S-DSM runtime: the main issue being that if a user code is currently running, and if it waits for a particular message from the S-DSM servers (eg a read or write acknowledgment message), then it has to postpone the treatment of this publish notification. The message is then pushed to an event pending list, to be later replayed. We propose the following task model, illustrated by the sequence diagram given in Figure 2.

A user task is defined by a mandatory *main* user function and several optional *handler* functions. The S-DSM runtime bootstraps on the main function. At the end of this function, it falls back to the builtin S-DSM client *loop* function that waits for incoming events such as publish notifications. If there are messages postponed in the event pending list, then they are locally replayed. If the task has no active chunk subscriptions, nor postponed messages in the pending list, then it effectively terminates.

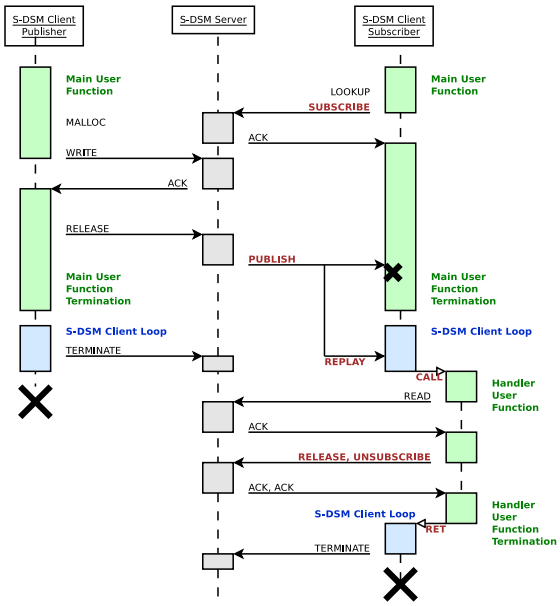


Fig. 2: *Sequence diagram for shared memory access and publish-subscribe events. S-DSM server can be replicated.*

The PS mechanism can work in two modes: 1) the notification mode only triggers a call to the user handler and 2) the push mode embeds a chunk update. This second mode can prefetch data if the user handler accesses the chunk. In the remaining of this paper we only consider the notification mode. The PS mechanism is not allowed to by-pass the consistency protocol and it is not possible to access the chunk outside the consistency scope. PS is a loosely coupled protocol and, for a given chunk, the only causal dependency between a PS notification and an access to the chunk within the handler is that the chunk version accessed is greater or equal to the chunk version that triggered the publish event. The notification and the shared access are not atomic, and several writes can occur on the chunk in-between. However, it is possible to implement a producer-consumer pattern with PS using two chunks as implemented in the application used for the experiments.

4 Experiments with an Heterogeneous Micro-server

Experiments have been conducted onto a RECS|Box heterogeneous micro-server. The form factor is a standard 1U rackable server that is composed by a backplane onto which it is possible to plug computing nodes. Figure 3 presents the micro-server configuration used for all experiments. Two Intel i7 nodes and two Arm-based nodes, the latter embedding 4 Cortex A15 processors each. We do not use the FPGA with Cortex A9 processor. The network is heterogeneous in terms of latencies and bandwidth. The ethernet interface of Cortex A15 processors is

implemented over USB with an internal switch within the node to connect the 4 processors. This explains the poor network performances when accessing these processors. Power consumption is monitored by contacting the remote control unit using the REST protocol.

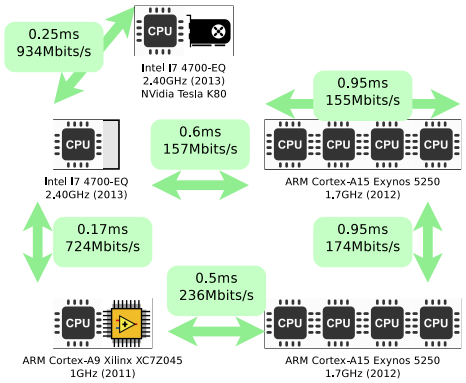


Fig. 3: *Heterogeneous Christmann RECS|Box Antares Microserver. Latencies are given by* Ping *and throughputs by* Iperf. *If not specified, we assume roughly the same performances as similar links.*

We consider a video processing application composed by one input task for video decoding and frame scheduling, N frame processing tasks, and one task for video encoding. The processing tasks perform edge detection using a $3 * 3$ stencil convolution, followed by line detection using a hough transform. While the convolution complexity is constant, the hough transform complexity is data-dependent: the complexity differs from one frame to another. Above a detection threshold, a pixel is represented as a sinusoid in the intermediate transformed representation. In this intermediate representation, above a second detection threshold, a pixel is represented as a line in the final output image. Both transform operations require the use of double-precision sinus and cosinus functions, which is quite demanding in terms of computational power. To illustrate the software heterogeneity, the processing task has been written in different technologies: sequential C, Pthread (4 threads), OpenMP, OpenCL and OpenCV (using the builtin OpenCV functions).

The application has been implemented using rendez-vous (RR for round-robin scheduling) and publish-subscribe (PS) synchronization functions over the S-DSM. Figure 4 represents the task interactions in both implementations. For each processing task, the input and output frames are stored into memory buffers that are allocated within the S-DSM chunks. The scheduling and buffer synchronization patterns differ in the RR and PS implementations: in the top half part of the figure, frames are written into the input buffers following a round-robin scheduling strategy implemented by the producer-consumer pattern based on rendez-vous synchronization. In the RR implementation, tasks will process the same number of frames (plus one extra frame depending on the video length). If there is a small parallelism degree -a small number of processing tasks- and if a task performs slowly, then this task becomes a bottleneck due to the round-robin strategy that will hang on this particular task until it has finished the job. In the second half part of the figure, the PS implementation, processing tasks are notified each time a new frame has been written into their associated input buffer. In

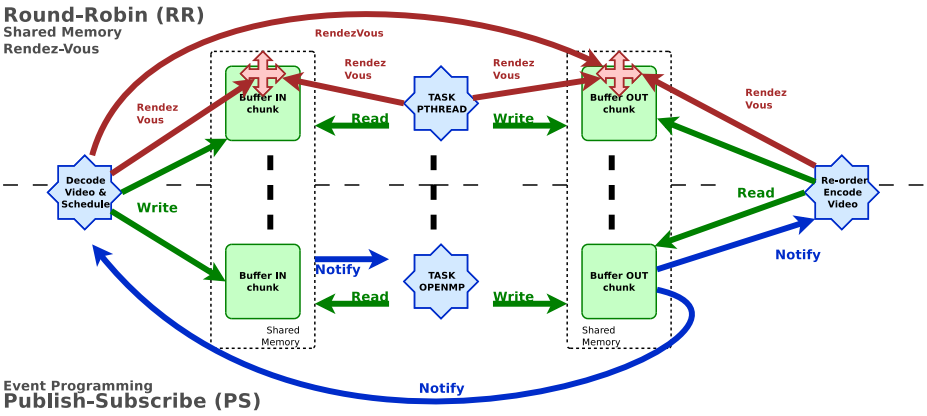


Fig. 4: Description of the video processing application. Top half part is the round-robin (RR) synchronization pattern while the second half part is the publish-subscribe (PS) pattern. The producer-consumer pattern is implemented using a mix of PS notification events and R/W shared memory access.

turn, both encoding and decoding tasks are notified each time a processed frame has been written to an output buffer. In that case, the encoding task reads this buffer and writes to the output, while the decoding task decodes the next frame and writes to the corresponding input buffer. This synchronization implements an eager scheduling strategy based on a mix of publish-subscribe notifications and S-DSM write events. The input is a 1-minute video file, with a total of 1730 frames and a resolution of 1280x720 pixels. Processing a frame using Pthread or OpenMP takes around 0.2s on a Core i7 (346s if we extrapolate to 1730 frames) and 0.9s on a Cortex A15 (1557s for 1730 frames). In OpenCV, it takes 0.05s on the Core i7 without external GPU (86.5s for 1730 frames). However, the OpenCV implementation provided by *libopencv* differs from other implementations and delivers quite different results.

Different configurations and mappings of the application are presented in Figure 5 and labeled from A to F. The heterogeneity is given for processing tasks only. For technical reasons, decoding and encoding tasks are implemented in OpenCV and are always deployed on Core i7. For each configuration, Figure 6 presents the number of frames processed by each task. The RR scheduling policy evenly distributes the workload among the tasks while the PS implementation reveals how tasks can process at different speeds depending on the hardware and the software choices.

PS performs better with software heterogeneity. In configuration A, a set of 4 processing tasks implemented with similar technologies -Pthread and OpenMP- are deployed on two i7 processors. Processing times are very close for both RR and PS implementations with a quite similar distribution of frames among the tasks. In configuration B, two tasks are implemented in OpenMP

	NODES		TASKS		HETEROGENEITY		TIME (s)		W
	i7	A15	Proc	Serv	Hardware	Software	RR	PS	
A	2	0	4	1	i7	Pthread OpenMP	220	218	58
B	2	0	4	1	i7	OpenMP OpenCV	177	153	58
C	1	8	8	1	A15	Sequential Pthread OpenMP	286	401	85
D	2	8	10	1	i7 A15	Pthread OpenMP	233	359	114
E	2	8	10	2	i7 A15	Pthread OpenMP	221	209	114
F	2	8	8	4	i7 A15	Pthread OpenMP	286	198	114

Fig. 5: Different configurations of the video processing application. For each configuration, the table gives the number of Intel Core i7 and Arm Cortex A15 processors used, the number of processing tasks (*Proc*), S-DSM data servers (*Serv*), the heterogeneity in terms of hardware and software, the total processing time for round-robin (RR) and publish-subscribe (PS) and the average instantaneous power consumption of the RECS|Box micro-server.

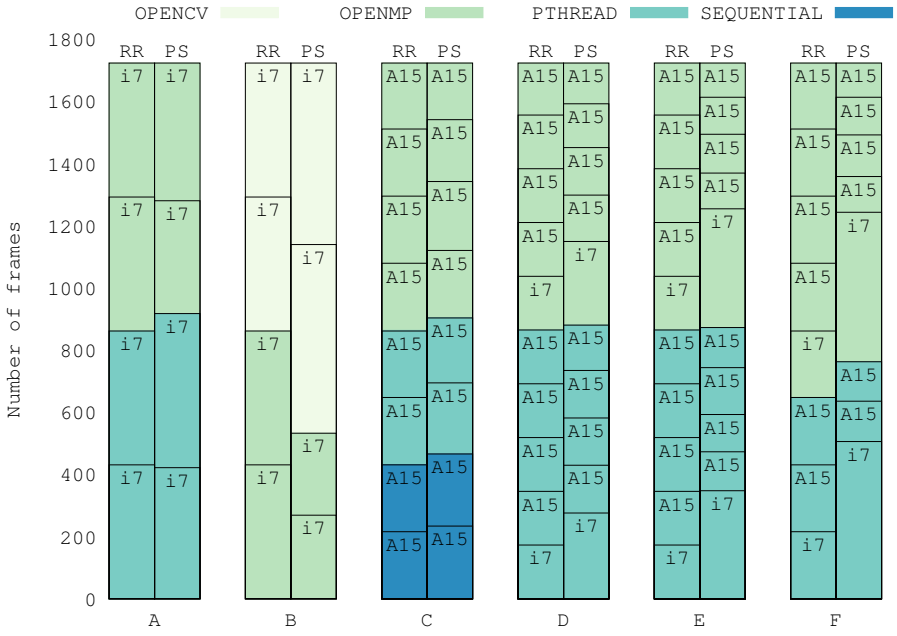


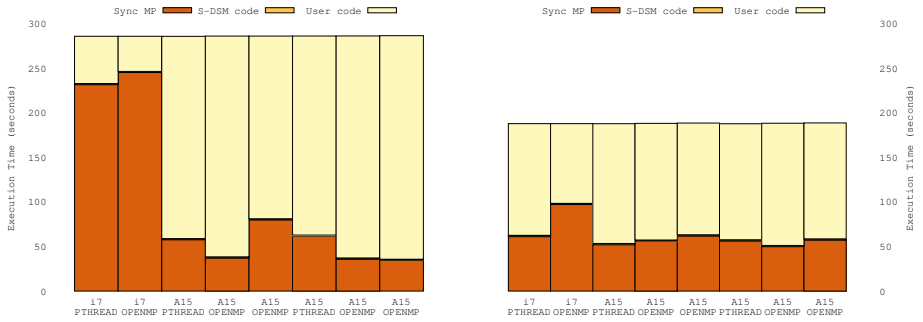
Fig. 6: Number of frames processed by each task for both RR and PS implementations. The software technology and the processor type into which the task is mapped are also displayed.

and two tasks in OpenCV, the latter being a faster code. All tasks are running on two i7 processors. The PS version performs better than RR by allowing the OpenCV tasks to process more frames than OpenMP.

RR performs better with low-performance network. In configuration C, 8 tasks are deployed over eight A15 processors. For technical reasons the decoding and encoding tasks require to be deployed on the i7 processor. Two processing tasks are implemented in a sequential C code, two in Pthread and four in OpenMP. While we were able to get the expected speedup between the sequential and the parallel implementations on Core i7 (almost 4 times faster), we observe that the sequential implementation performs slightly better than the parallel ones onto Cortex A15. However, the PS implementation is not adapted to this configuration, due to the poor network capabilities of our testbed, especially considering the latencies with the Arm baseboards. In that case, the RR implementation provides a better use of the network by avoiding bursts of small messages. In configuration D, 10 processing tasks are deployed over the two i7 and eight A15 processors. The PS implementation distributes more frames to the tasks running on the i7 processors than to the A15. However, as for configuration C, it does not perform well compared to the RR implementation. The S-DSM is deployed using one server and all memory access requests and publish-subscribe notifications are converging to the same i7 node.

PS benefits from distributed metadata management. Configurations E and F respectively use 2 and 4 S-DSM servers to manage data and metadata. In configuration E, S-DSM servers are deployed over the two i7 processors. The PS implementation largely benefits from this configuration, going from 359s with one server to 209s with two servers. In configuration F, two more S-DSM servers are deployed, with a total of one server per baseboard. While this approach involves more communications between servers and slows down the RR implementation, it is well adapted to the PS event-based implementation which performs slightly better than configuration E. This is quite a new result for the proposed S-DSM for which it is rarely worth to distribute the data and metadata management among different servers at this scale when using regular shared access primitives. Instead, we notice that the publish-subscribe mechanism requires a better load balance of events if the network is slow, even for small configurations.

PS balances idle times among tasks. Figure 7 presents the execution time per task for both RR and PS implementations using configuration F. For each task the time is decomposed into three parts: 1) the *Sync MP* corresponds to the time spent in the message passing receive primitive. This mainly happens while waiting for a ack message when accessing the shared memory, a rendez-vous release or a publish-subscribe notification. For HPC applications, this *Sync MP* time should be avoided because it reveals that the task is waiting rather than processing data. 2) The *S-DSM code* time corresponds to the local S-DSM data management. It is usually not significant, with less than 0.7% of the task total execution time in this example. 3) The *User code* corresponds to the time spent



(a) RR (b) PS

Fig. 7: S-DSM and application time per task for both RR and PS implementations using configuration F. Figures 7a and 7b use the same vertical scale.

in the user code execution, excluding S-DSM calls. In the RR implementation, tasks running on i7 processors spend up to 86% of the time waiting on S-DSM rendez-vous, compared to 12% to 28% for tasks running on the A15 processors. In the PS implementation, while drastically decreasing the total computing time (more than 1.5 faster), all tasks spend between 27% and 51% of their time waiting for the next S-DSM events.

5 Related Works

Distributed shared memory and publish-subscribe systems have been widely studied in the literature for the past decades. Most of the DSM systems, starting with IVY [12] only provide mechanisms for implementing the shared memory paradigm. Cache coherence protocols based on write-invalidate or write-update policies such as MESI [10] are quite close to the publish-subscribe pattern: memories that host a copy of the data are in fact subscribing to its modifications. However, the event is defined by the sole protocol, whether it is a change of the data status in the metadata structure or the update of the data in the memory. There is no third-party application nor user code that can be called on such event. Event-based programming can be used to implement a DSM, as proposed in this system [13], which describes a DSM implemented using the Java event-based distributed system. Our contribution is to implement an event-based distributed system on top of the S-DSM, which is the opposite approach. Publish-subscribe systems have been successfully used for GUIs, internet services, multicasting in mobile networks [6,2] and managing immutable shared data in peer-to-peer (P2P) systems [11,15]. The PS programming paradigm is quite different from shared memory and as far as we know, there is no such system that merge both paradigms. These two paradigms are shaped for very different computing

contexts: homogeneous reliable computing nodes with HPC code for DSM and heterogeneous volatile devices with service-oriented code for PS. With the emergence of heterogeneous systems mixing both HPC and event-based applications, we think that our contribution can ease the programmability of the platform. One example is the integration of the event-based system SOME/IP [19] within the AUTOSAR specification standard for future automotive systems, in which heterogeneous computing nodes will have to deal with both HPC applications for vehicle guidance and service-oriented applications for entertainment. The design of the video processing application used in this paper is very close to a dataflow application, in which a set of tasks communicate using explicit channels. Some dataflow runtimes include StarPU [5] designed for heterogeneous architectures with a complementary S-DSM used for internal data management. However, the programming model exposed to the user is pure dataflow, and not a mix of both shared memory and PS paradigms as proposed in this paper.

6 Conclusion

Heterogeneous distributed architectures are expected to be deployed in future technological systems for data centers, industry and automotive. Each application field relies on historical programming models and we propose to merge both shared memory with publish-subscribe models, building a bridge between very different application contexts. We found the underlying mechanisms very similar, leading to a quite straightforward integration. This work contributes with 1) a programming model that merges shared memory and publish-subscribe, 2) a task model that bootstraps on the main user code and terminates with the event-based model and 3) experiments on a heterogeneous micro-server. The experiments show that the choice between shared memory and PS should be made according to the application configuration and the execution platform.

Acknowledgments. This work received support from the H2020-ICT-2015 European Project M2DC - Modular Microserver Datacentre - under Grant Agreement number 688201.

References

1. Amza, C., Cox, A.L., Dwarkadas, S., Keleher, P., Lu, H., Rajamony, R., Yu, W., Zwaenepoel, W.: TreadMarks: Shared memory computing on networks of workstations. IEEE Computer **29**(2), 18–28 (Feb 1996)
2. Anceaume, E., Datta, A.K., Gradinariu, M., Simon, G.: Publish/subscribe scheme for mobile networks. In: Proceedings of the Second ACM International Workshop on Principles of Mobile Computing. pp. 74–81. POMC '02, ACM, New York, NY, USA (2002)
3. Antoniu, G., Bougé, L.: Dsm-pm2: A portable implementation platform for multithreaded dsm consistency protocols. In: Proceedings of the 6th International Workshop on High-Level Parallel Programming Models and Supportive Environments. pp. 55–70. HIPS '01, Springer-Verlag, London, UK, UK (2001)

4. Antoniu, G., Bougé, L., Jan, M.: JuxMem: an adaptive supportive platform for data-sharing on the grid. Scalable Computing: Practice and Experience (SCPE) **6**(3), 45–55 (Nov 2005)
5. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.A.: StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009 **23**, 187–198 (Feb 2011)
6. Banavar, G., Chandra, T., Mukherjee, B., Nagarajarao, J., Strom, R.E., Sturman, D.C.: An efficient multicast protocol for content-based publish-subscribe systems. In: Proceedings. 19th IEEE International Conference on Distributed Computing Systems (Cat. No.99CB37003). pp. 262–272 (1999)
7. Bershad, B.N., Zekauskas, M.J., Sawdon, W.A.: The Midway distributed shared memory system. In: Proceedings of the 38th IEEE International Computer Conference (COMPCON Spring '93). pp. 528–537. Los Alamitos, CA (Feb 1993)
8. Bisiani, R., Forin, A.: Multilanguage parallel programming of heterogeneous machines. IEEE Trans. Comput. **37**(8), 930–945 (Aug 1988)
9. Cudennec, L.: Software-distributed shared memory over heterogeneous micro-server architecture. In: Euro-Par 2017: Parallel Processing Workshops. pp. 366–377. Springer International Publishing (2018)
10. Culler, D., Singh, J., Gupta, A.: Parallel Computer Architecture: A Hardware/-Software Approach. Morgan Kaufmann, 1st edn. (1998), the Morgan Kaufmann Series in Computer Architecture and Design
11. Ginzler, T.: A robust and scalable peer-to-peer publish/subscribe mechanism. In: 2012 Military Communications and Information Systems Conference (MCC). pp. 1–6 (Oct 2012)
12. Li, K.: IVY: a shared virtual memory system for parallel computing. In: Proc. 1988 Intl. Conf. on Parallel Processing. pp. 94–101. University Park, PA, USA (Aug 1988)
13. Mazzucco, M., Morgan, G., Panzieri, F., Sharp, C.: Engineering distributed shared memory middleware for java. In: Meersman, R., Dillon, T., Herrero, P. (eds.) On the Move to Meaningful Internet Systems: OTM 2009. pp. 531–548. Springer Berlin Heidelberg, Berlin, Heidelberg (2009)
14. Morin, C., Kermarrec, A.M., Banatre, M., Gefflaut, A.: An efficient and scalable approach for implementing fault-tolerant dsm architectures. IEEE Transactions on Computers **49**(5), 414–430 (May 2000)
15. Nakayama, H., Duolikun, D., Enokido, T., Takizawa, M.: A p2p model of publish/subscribe systems. In: 2014 Ninth International Conference on Broadband and Wireless Computing, Communication and Applications. pp. 383–388 (Nov 2014)
16. Nelson, J., Holt, B., Myers, B., Briggs, P., Ceze, L., Kahan, S., Oskin, M.: Latency-tolerant software distributed shared memory. In: 2015 USENIX Annual Technical Conference (USENIX ATC 15). pp. 291–305. USENIX Association, Santa Clara, CA (2015)
17. Pinheiro, E., Chen, D., Dwarkadas, H., Parthasarathy, S., Scott, M.: S-dsm for heterogeneous machine architectures (07 2000)
18. Santo, M.D., Ranaldo, N., Sementa, C., Zimeo, E.: Software distributed shared memory with transactional coherence - a software engine to run transactional shared-memory parallel applications on clusters. In: 2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing. pp. 175–179 (Feb 2010)
19. Völker, L.: Some/ip die middleware für ethernet-basierte kommunikation. Hanser automotive networks (Nov 2013)