Proceedings of the 2020 IEEE/SICE
International Symposium on System Integration
Honolulu, Hawaii, USA, January 12-15, 2020

Toward Distributed Streaming Data Sharing Manager for Autonomous Robot Control

Hiroaki Fukuda[1], Ryota Gunji[2], Tadahiro Hasegawa[3], Paul Leger[4] and Ismael Figueroa[5]

Abstract— **Using robots is demanding for supporting our lives and/or covering works that are not suitable for human beings. The robot software implementation requires a variety of knowledge and experiences. Thereby developing cost for such software systems is now increasing. Middleware systems such as Robot Operating System (ROS) are being developed to decrease such cost and widely used. Streaming data Sharing Manager (SSM) is one of such middleware systems that allow developers to write and read sensor data with timestamps using a common PC. This feature enables developers to control a robot by taking account of measured time. This control is important because using multiple sensor data with different timestamp cannot allow developers to control a robot correctly. SSM assumes that only one PC is used to control a robot, therefore if it exists a process that consumes much CPU resource, other processes cannot finish their assumed deadlines, leading to the unexpected behavior of a robot.**

This paper proposes an architecture called Distributed Streaming data Sharing Manager (DSSM) that enables to distributing each process on existing SSM to different PCs. We investigate the current architecture and behavior of SSM, then propose a new architecture that can achieve our goal. Finally, we apply DSSM to an existing SSM based robot control system that autonomously controls an unmanned vehicle, then confirm the effectiveness of DSSM by measuring the resource usages.

I. Introduction

Robotics is a hot topic which will support our lives [5] [6] and/or cover some tasks that are not suitable for human beings shortly. A robot consists of several sensors, actuators, thereby developing a software system that controls the robot (we call this software as a robot control system in this paper) needs a variety of field's knowledge and experiences. Middleware systems such as Robot Operating System (ROS) [7] [8] and/or RT-Middleware (RTM) [3] [9] [4] are now being developed to decrease the developing cost and are widely used nowadays [3] [7]. Streaming data Sharing Manager (SSM) [1] is one of the middleware systems for developing a robot control system using a common PC. In a robot control system, a single process is in charge of measuring data using a sensor or handling an actuator, meaning that a robot control system consists of multi processes. Suppose that we need to use multiple kinds of sensors and actuators to implement a robot control system. In this robot control system, we will create a process that mainly controls the entire system (we call this process as a main process), meaning that the main process needs interprocess communications (shortly IPCs) among other processes that are in charge of measuring sensor data because the main process will decide the entire behavior of the robot referring to multiple sensor data. In this situation, the timestamps of measured sensor data are important because the main process cannot control the entire system correctly if it uses multiple sensor data with different timestamps.

SSM provides developers with useful APIs that hide complexities regarding timestamps from developers. A robot control system that uses SSM is also running as multiple processes in a PC, thereby SSM provides shared memories for the IPCs among processes Because of this restriction, a process with a heavy load that will consume CPU power might affect other processes, leading to the delay or an unexpected control of a robot.

This paper proposes an SSM based new middleware system called Distributed Streaming data Sharing Manager (DSSM) that enables to distributing each process on different PCs while existing software systems that use SSM do not need to be modified as much as possible by designing DSSM's architecture. DSSM introduces TCP/IP communications to divide multiple processes, that were running on a PC, into multiple PCs, then provides similar APIs that SSM has provided already with developers. Also, we apply DSSM to an existing SSM based robot control system that autonomously controls an unmanned vehicle, then observe its behavior and measure the resource usages to verify the prototype implementation of DSSM.

The remainder of this paper is organized as follows. Section II describes the SSM architecture and Section III explains the DSSM architecture and main components. Section IV presents an experiment and results then show the efficiency of DSSM compared to SSM. Section V discusses related work and Section VI finally concludes this paper with future work.

II. SSM: Streaming data Sharing Manager

In this section, we describe the background of SSM, which means the difficulties of controlling autonomous robots and requirements. Then, we briefly explain the architecture of SSM.

[1]Hiroaki Fukuda is with Department of Computer Science and Engineering, Shibaura Institute of Technology, Japan hiroaki@shibaura-it.ac.jp

[2]Ryota Gunji is with Graduate School of Electrical Engineerning and Computer Science, Shibaura Institute of Technology, Japan ma19030@shibaura-it.ac.jp

[3]Tadahiro Hasegawa is with Department of Electrical Engineerning, Shibaura Institute of Technology, Japan thase@shibaura-it.ac.jp

[4]Paul Leger is with Escuela de Ingenieía, Universidad Católica del Norte, Chile pleger@ucn.cl

[5]Ismael Figueroa is with Escuela de Auditoría, Universidad de Valparaíso, Chile ismael.figueroa@uv.cl

978-1-7281-6667-4/20/$31.00 ©2020 IEEE

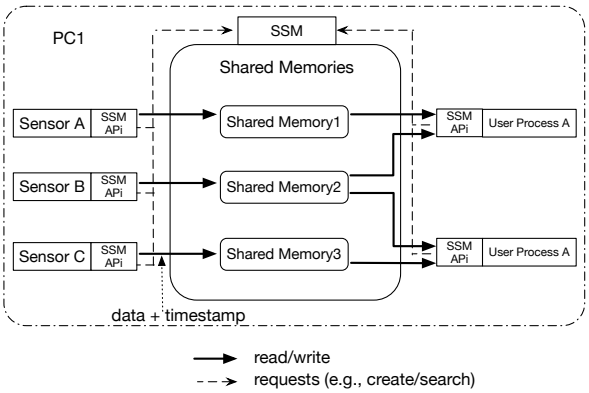

Fig. 1. The usage of SSM

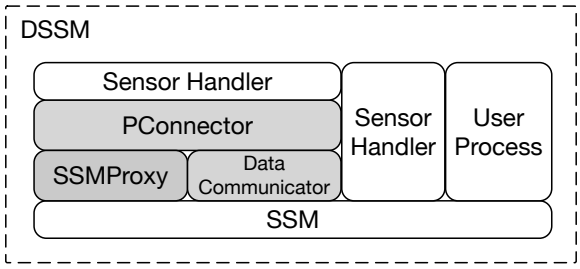

Fig. 2. DSSM Architecture

A. Difficulties of controlling autonomous robots

An autonomous robot generally uses a variety of sensors that measure several values such as velocities, directions and the distance to objects. A robot control system detects the environment and reacts to avoid unexpected behaviors. Besides, a complicated software system should consist of several modules because of the customizable and extensible requirements. Suppose that we implement a robot control system that controls an unmanned vehicle. The vehicle should stop as soon as it detects a human being in front of it. The robot control system may use a camera to detect a human being, then stop the engine or put on the brake as reactions. In this scenario, we will implement this robot control system using two modules: The first module uses a camera to detect human beings then write the result (e.g., the camera detects human beings or not). The second module reads the result and reacts (e.g., stop the engine). These modules are usually run as different processes. In this scenario, as we mentioned in Section I, timestamps are also important in addition to raw sensor data. This is because, in this scenario, the first module periodically writes the result, therefore the second module has to choose the adequate one from the results written by the first module. Therefore it is desirable to manage data measured by sensors with timestamps. SSM is a middleware system that hides complexities of managing timestamps and provides APIs that allow developers to write and read data using timestamps.

B. Architecture of SSM

We show an usage of SSM in Figure 1. SSM uses shared memories for IPCs because it assumes that all modules are run on a single PC even though each module will be run as different processes. SSM provides APIs, called *SSMAPi*, to handle the shared memory (e.g., create, read, write and delete). With these SSMAPis, timestamps are implicitly added to each measured sensor data. In addition, we call a program that measures sensor data as *sensor handler*. In Figure 1, "Sensor A", "Sensor B" and "Sensor C" are sensor handlers, then they use SSMAPi inside. In SSM, a sensor handler firstly sends a request to SSM in order to create a shared memory. The sensor handler also sends an identifier to distinguish the shared memory it creates and uses. When the SSM has successfully created the shared memory, it returns the pointer of the shared memory to the corresponding sensor handler. The sensor handler can directly write any data to the shared memory via SSMAPi.

Besides, a program that controls a robot (we call *User Process* in Figure 1) needs to read data stored in shared memories. The program sends a lookup request for finding a certain shared memory to SSM using an identifier, then SSM returns the pointer of corresponding shared memory. Once the program obtains the pointer, it can directly read data from the shared pointer by specifying timestamps via SSMAPi. The details of complicated processes (e.g., creating shared memories) are hidden by SSM, therefore developers can easily use these functions. Moreover, SSMAPi is not only be used for creating new shared memory but also be used for other requests to SSM such as terminating or searching existing shared memory. SSM uses a message queue system as an IPC and handles more than one request at a time.

SSM was designed as a middleware system with which developers can implement a robot control system using multiple processes because of several requirements such as customizability and extensibility while minimizing the delay of IPCs, result in using shared memories. On the other hand, using shared memories makes it impossible to run each process on different PCs. As a result, a process that intensively consumes CPU power such as image processing affects the entire behavior of the system because these processes generally use the same CPU. In fact, we developed a robot control system that controls an unmanned vehicle using SSM. This software system uses a camera to detect objects for avoiding crashes, then using camera intensively consumes CPU power. As a result, we could not stop the vehicle before hitting an object even though the camera detected it. To avoid this situation, we decreased the speed of the vehicle, which was a compromised solution.

III. DSSM: Distributed data Sharing Manager

In this section, we explain the details of DSSM that is our proposal in this paper. We firstly illustrate the architecture of DSSM, then explain the main components in DSSM. We lastly describe the synchronized problem that is a side effect introducing DSSM, and the corresponding solution.

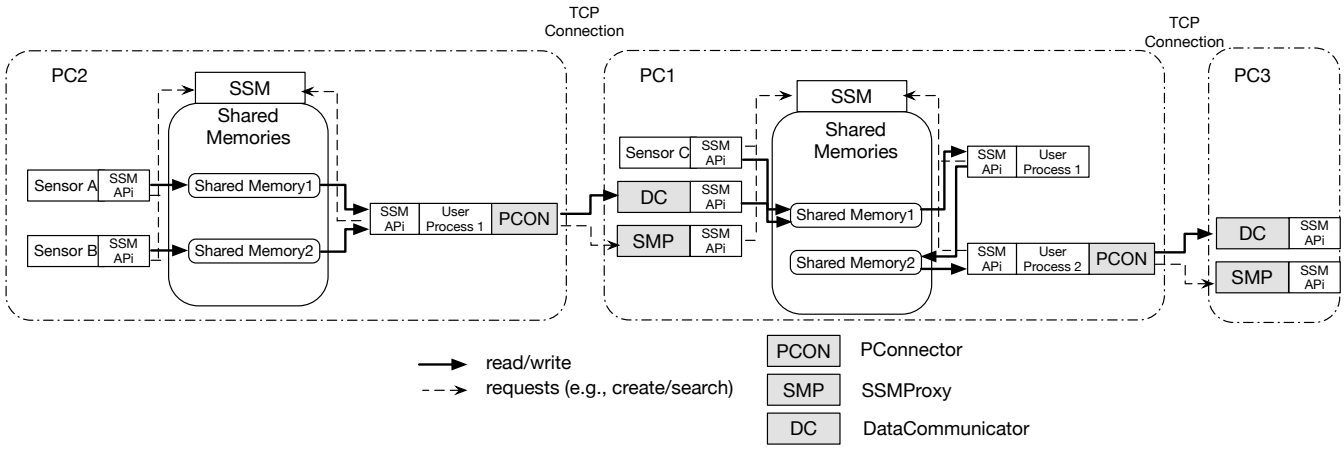

Fig. 3. Use case of DSSM

A. Architecture of DSSM

In Figure 2, we show the new architecture that enables to distributing existing processes to different PCs while minimizing changes of existing programs. We use the current architecture of SSM in which all programs are running as processes. We will give an additional IPC using network without modifying the core of SSM. We added the following three main components:

SSMProxy: This component will be run on the PC where an SSM is running. SSMProxy will accept requests from clients (i.e., PConnectors), and they behave as sensor handlers in existing SSM, meaning that SSM will accept requests from SSMProxy directly.

DataCommunicator: This component will be instantiated when a shared memory is created in SSM. From a shared memory viewpoint, this component directly writes/reads data to/from the shared memory.

PConnector: This component will be used sensor handlers that write measured sensor data to SSM which is running on the different PC. PConnector directly sends/receives data to/from SSMProxy using TCP connections.

B. Behavior of each component

We show the use case of DSSM in Figure 3. In DSSM, SSMProxy on the same PC where SSM runs as an isolated process. When SSMProxy accepts a request from a PConnector, it creates a child process to use fork system call, then the child process will handle all requests sent from the corresponding PConnector. This is because the original SSM handles requests from more than one sensor handlers at a time, then we will keep this architecture in our new architecture using network. The SSMProxy uses SSMAPi to send requests to SSM such as creating or deleting shared memories, meaning that an SSMProxy seems to be a sensor handler from SSM viewpoint.

On the other hand, a sensor handler, which needs to access a shared memory across the network, utilizes PConnector in DSSM. The implementation of a sensor handler needs to be changed from SSMAPi to using PConnector. Even though this change will be required, we will minimize the changes by applying similar interfaces provided by the current SSMAPi. Because of this, developers will be able to get familiar with PConnector soon. PConnector will be in charge of communicating with SSMProxy and DataCommunicator using TCP connection. As shown in Figure 3, we can utilize SSMAPi and PConnector at the same time like Sensor handler2 within PC2, meaning that this allows developers to design a software system more flexibility. For example, this enables to write processed data to other PC using PConnector after writing to raw data to local shared memory using SSMAPi.

TABLE I
TIME DELAY BETWEEN NTPSERVER AND PC USING NTPD

	Average	Max	Min	Standard Deviation
offset(ms)	1.41	5.93	0.015	1.20

When an SSMProxy receives a request which offers to create a shared memory, the SSMProxy sends a request to SSM running on the same PC, then it also creates an instance of DataCommunicator to handle write/read data from a PConnector. The DataCommunicator will be run as a different thread. This is because a sensor handler in the original SSM can write data to more than one shared memory. A single TCP connection between a PConnector and the corresponding SSMProxy cannot provide this behavior. Thereby every DataCommunicator will open a socket to make a connection to a PConnector, then write data sent from the PConnector to the shared memory or send the data from the shared memory to the PConnector.

With these three additional components, we can run existing software systems that use SSM as distributed manner while minimizing modifications.

C. Time synchronize problem and a solution

As we mentioned in Section II-A, controlling autonomous robots requires not only raw sensor data but also their

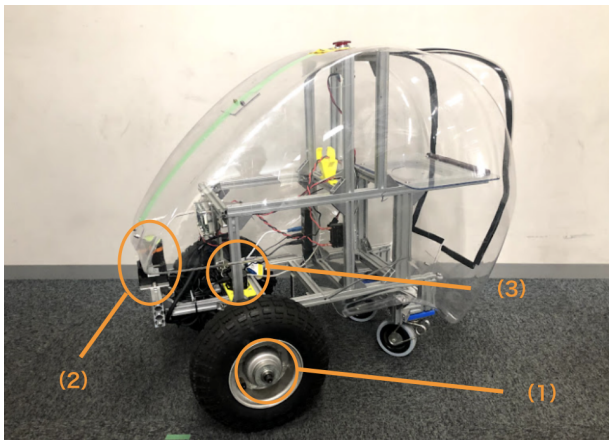

Fig. 4. Shape of our autonomous controlled robot and equipped sensors.

Fig. 5. Map and trajectory of the unmanned vehicle with DSSM

measured timestamps. SSM assumes that all modules of a software system using SSM are running on a single PC. Thereby all modules share the same clock, meaning that we do not need any concerns about time synchronization. On the other hand, in DSSM, these modules are distributed in different PCs in which each PC has its own clock. There are several proposals to handle this problem [10] [11] [12]. Then, we use Network Time Protocol (NTP) [11] to tackle with this problem concering the balance of a requirement and delay. We run a NTP daemon (NTPD) in a PC (PC1 in Figure 3), then other PCs synchronize the time to the PC. Using two PCs, we measured the offset using NTPD as an average of 140 evaluations of the offset value. Table I shows the result where the maximum offset was less than 6(ms) and the average was about 1.5(ms). As we explain in Section IV, these results satisfy the requirement of controlling an autonomous robot in this paper.

IV. Experiment and Evaluation

As we mentioned in Section II-B, we developed a robot control system that controls an unmanned vehicle using SSM. We show the vehicle in Figure 4. This vehicle mainly equips three sensors: (1)Encoders to measure wheel odometries that are used to estimate the position of the vehicle. (2) Light Detection and Ranging (LIDAR) that measures the distance to objects to estimate a position of the vehicle using the scan matching, and also avoid crashes. (3) Inertial Measurement Unit (IMU) that measures an angular velocity to estimate a yaw angle of the vehicle.

We run 5 processes to control the vehicle: three processes are for measuring sensor data, one process is for estimating the position of the vehicle using measured sensor data and the main process controls the vehicle reacting the estimated position.

Figure 5 shows the map used in this experiment and the trajectory of the vehicle. The robot autonomously runs from *START* to *GOAL* while rounding the circle object. In Figure 5, (1) represents the trajectory and (2) shows the estimated position of objects (i.e., walls in this case) that correctly matched the map. In this experiment, we use three PCs (PC1, PC2, PC3) with DSSM. We attached the encoder and IMU to the PC1 while LIDER was attached to PC2. The viewer was running on PC3 for us to confirm the behavior of the vehicle. Besides, as shown in Figure 5, we confirmed that this vehicle could be controlled by distributed manner without any problems. Note that, in the current implementation, this robot control system adjusts the vehicle's position and direction every 25ms. Therefore using NTPD can satisfy this requirement.

TABLE II
Comparison of CPU usage on PC1

	user	nice	system	iowait	steal	idle
PC1 SSM	37.55	0.08	2.53	0.29	0.00	59.55
PC1 DSSM	31.08	0.15	4.81	1.19	0.00	62.78

On the other hand, we also measure the average CPU usage of PC1 while this experiment and compare it to the case of SSM in which all processes are running on PC1 for comparison. PC1 is in charge of controlling the vehicle while PC2 is in charge of estimating the vehicle's position and direction. PC3 is in charge of showing the status of the vehicle using three sensor data. Sensor data are measured on PC1 and PC2.

As an evaluation, we confirmed the behavior of vehicle with DSSM, in which we could not find any big differences of behaviors using SSM with visual observation, meaning that DSSM was working without any problems. Besides, we show the CPU usage of PC1 in Table II. On the whole, DSSM could decrease CPU usage of PC1 compared to SSM. However, two elements such as *system* and *iowait* in DSSM were increased. This is because DSSM uses additional I/O (i.e., network) and system calls compared to SSM. Based on these results, the effect of DSSM seems to be limited, however, we do not use a camera and/or other sensors that might consume CPU intensively. We need to extend these experiments to show the effectiveness of DSSM in the future.

V. Related Work

ROS [7] [8] is a middleware system that adopts the publish/subscribe model. ROS processes are represented as nodes and each node constitutes a graph structure, meaning that each node can directly connect each other. In ROS, a node uses TCP connections for IPCs. Thereby if two nodes are running on the same PC, they must connect using TCP connections, leading to a certain delay. Meanwhile, ROS provides a method to share data in a single process, resulting in minimizing the delay. In this case, however, all functions that are required to control a robot should be run in a single process as threads, leading to tightly-coupled relations among them, that is not suitable for modularity view point. Besides, ROS is not in charge of timestamps, therefore developers need to handle them by themselves.

RTM [3] [9] is also a middleware system. Similar to ROS, in RTM, processes are represented as RT-Components and each RT-Component communicates using CORBA. Since CORBA uses TCP connections, using RTM will encounter a certain delay because of the same reason as ROS, RTM is also not in charge of timestamps of measured data.

Compared to these middleware systems, as shown in Figure 3, DSSM allows developers to combine two types of IPCs: shared memories and TCP connections, as they prefer. For example, if two processes require high-speed communications between them, developers can choose shared memories in a single PC. On the other hand, if a process will intensively consume resources, developers can easily separate the process on a different PC and choose TCP communications. Moreover, DSSM is in charge of timestamps for measured sensor data. Besides, if developers already use SSM as a middleware system, they can easily introduce DSSM will small changes.

VI. Conclusions

With the demand of robots, middleware systems are now developed and used to decrease the development cost of software systems which control robots. SSM is one of such middleware systems that provide a method to write/read data with timestamps. The current SSM assumes that all programs that control a robot run in a single PC as different processes. Thereby one heavy program (process) that intensively consumes CPU resource affects other programs, leading to unexpected results of the robot control.

This paper proposes DSSM that makes it possible to run the programs in different PCs. DSSM provides two types of IPCs: shared memories and TCP connections. Therefore developers can choose and/or combine these IPCs based on the requirements that are new points of DSSM different from existing middleware systems such as ROS. We consider and propose three additional core components to achieve this goal while existing software systems need not be modified as much as possible. We give a prototype implementation of DSSM and apply it to the existing software system that autonomously controls an unmanned vehicle. We conducted a basic experiment and the result of which shows DSSM works fine and slightly decreases CPU usage.

Regarding future work, we should conduct more experiments with real use cases and verify the effectiveness of DSSM. Also, we should add more useful tools such as monitoring systems for developers to use DSSM easily.

References

[1] Streaming data Sharing Manager (SSM), https://www.roboken.iit.tsukuba.ac.jp/platform/wiki/ssm/index, 2019/11/1 accessed.

[2] Tomoyoshi Eda, Tadahiro Hasegawa, Shingo Nakamura and Shinichi Yuta, Development of Autonomous Mobile Robot “MML-05” Based on i-Cart Mini for Tsukuba Challenge 2015, Journal of Robotics and Mechatronics, Vol.28 No.4, pp.461-469, 2016.

[3] N. Ando, T. Suehiro, K. Kitagaki, T. Kotoku, and Woo-Keun Yoon. 2005. RT-middleware: distributed component middleware for RT (robot technology). In 2005 IEEE/RSJ International Conference on Intelligent Robots and Systems. 3933-3938.

[4] Noriaki Ando, Shinji Kurihara, Geoffrey Biggs, Takeshi Sakamoto, Hiroyuki Nakamoto, ”Software Deployment Infrastructure for Component Based RT-Systems”, Journal of Robotics and Mechatronics, Vol.23, No.3, pp.350-359, 2011.06

[5] Michael Goodrich and Alan Schultz. 2007. Human-Robot Interaction: A Survey. Foundations and Trends in Human-Computer Interaction 1 (01 2007), 203–275.

[6] James Kramer and Matthias Scheutz. 2007. Development environments for autonomous mobile robots: A survey. Autonomous Robots 22, 2 (01 Feb 2007), 101–132.

[7] Morgan Quigley, Ken Conley, Brian P. Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y. Ng. 2009. ROS: an open-source Robot Operating System. In ICRA Workshop on Open Source Software.

[8] Yuya Maruyama, Shinpei Kato and Takuya Azumi, Exploring the Performance of ROS2, In Proceedings of the 13th International Conference on Embedded Software, 2016.

[9] N. Ando, T. Suehiro, K. Kitagaki, T. Kotoku and Woo-Keun Yoon, ”Composite component framework for RT-middleware (robot technology middleware),” Proceedings, 2005 IEEE/ASME International Conference on Advanced Intelligent Mechatronics., Monterey, CA, 2005, pp. 1330-1335.

[10] Leslie Lamport. 1978. Time, clocks, and the ordering of events in a distributed system. Commun. ACM 21, 7 (July 1978), 558-565.

[11] David L. Mills, Computer Network Time Synchronization: The Network Time Protocol, CRC Press, 2006.

[12] Mukesh Singhal and Ajay Kshemkalyani. 1992. An efficient implementation of vector clocks. Inf. Process. Lett. 43, 1 (August 1992), 47-52.