Systemic Support for Transaction-Based Spatial-Temporal Programming of Mobile Robot Swarms

Daniel Graff, Daniel Röhrig, Reinhardt Karnapke
Communication and Operating Systems Group
Technische Universität Berlin
10587 Berlin, Germany
Email: {daniel.graff, daniel.roehrig, karnapke}@tu-berlin.de

***Abstract*—In this paper, we present an approach to support transaction-based spatial-temporal programming of mobile robot swarms on a systemic level. We introduce a programming model for swarms of mobile robots. Swarm applications consist of concurrent, distributed and context-aware actions. We provide distributed transactions in order to guarantee atomic execution of a set of dependent actions. We distinguish between schedulability and executability of a set of actions. In order to guarantee executability of a distributed transaction of spatial-temporal actions, we present the concept of path alternatives and a time-based two-phase commit protocol in order to assure consistency. We show the feasibility of our approach by a proof-of-concept.**

I. Introduction

There is a rapid increase in the number of electronic devices ranging from deeply embedded sensors and actuators over wearable devices to fully autonomous robots. These devices form distributed sensing and actuating platforms that are highly interconnected. Based on different device manufacturers and system developers, a variety of different hardware, different system software with different system interfaces exists, exposing heterogeneity.

There are numerous applications that require access to specialized sensors and actuators in order to gather data (e.g., wind speed, temperature, humidity, seismographic activity, ..). Examples include: Traffic management systems that require access to traffic data and weather data. Flood prediction systems that require access to weather data and data about the water level of a certain river section. Long-term bridge monitoring systems that require access to traffic data, weather data and seismographic data. Robot-based exploration or observation systems that require access to specialized cameras and probes.

All these applications have in common that they require context awareness concerning physical space and time since their functional outcome heavily depends on these context parameters. Considering the above mentioned applications, there is a noticeable fraction of the applications that require similar or even the same sensors and actuators. Current approaches tend to set up their own infrastructure (interconnected hardware components) that are perfectly tailored to solely run one application.

We propose to share hardware resources and execute applications on the same physical infrastructure. An advantage of sharing resources in a timely manner is a significant reduction of deployment and maintenance costs. In order to achieve this, we consider the sum of all those heterogeneous devices as one emerging system (the swarm) and build a swarm operating system on top of it that manages and coordinates its resources. Applications are scheduled and managed by the swarm operating system. In order to allow multi-program operation, we use the concept of virtualization on resource level. Application developers profit from our approach since the operating system hides all heterogeneity, is responsible for resource management, synchronization and coordination and provides one clear defined interface against which applications are implemented. The operating system schedules all applications according to their needs in space and time.

This paper is structured as follows: Section II introduces a programming model for the swarm and describes how swarm applications are developed. Swarm applications consist of distributed transactions whose elements are spatial-temporal actions (Section III). Section IV gives an evaluation while Section V gives an overview about related work in this field and Section VI summarizes the paper.

II. Swarm Programming Model

The programming model describes how applications are implemented against the system. The model supports to make applications context-aware to physical space and time which is a necessity in cyber-physical systems while maintaining important transparencies in distributed systems such as distribution-, motion and concurrency transparency.

We differentiate between the *application model* and the *system model* as depicted in Figure 1: The system model describes the capabilities and properties that the system provides. Each device has its own description of capabilities and properties that contributes to the global system. A capability description can be for instance {*Camera: Resolution: 1024x768, Color: RGB, FPS: 25*} or {*Temperature-Sensor: Range: [-50, +50], Resolution: 0.1*}. A property describes the devices geometry, e.g., the shape of a robot and its maximum velocity. Stationary devices have a velocity of 0.

In order to use the capabilities of the system, we use the concept of the *lib / driver* model. All capabilities are operated by a dedicated driver that accesses the hardware and delivers the requested data. On the software engineering side, we provide a library that makes capabilities accessible for applications. The library communicates with the driver.

978-1-4673-6773-8/15/$31.00 ©2015 IEEE

The application model states how applications are developed. The model consists of three elements: *SwarmApplication*, *SwarmAction* and *ActionSuite*. For simplicity, we refer in the following only to applications, actions and suites. The application consists of loosely coupled actions. Each action is a library function that is executed on a certain capability. An action can be for instance *take picture* or *measure temperature*. Each action can be restricted in space and time by spatial-temporal constraints. Constraints can be absolute or relative to each other. Thus, an action could be *measure temperature* at location X_1 and at time t_1. Due to this programming semantic, the programmer is able to specify arbitrary compositions of actions, e.g., perform a uniformly distributed monitoring of an area at the same time t and repeat this every time interval t_I. In this example it becomes clear that, given the same time constraint, it is not possible to execute the actions on a single device since all location constraints for the actions differ. Thus, the amount of actions are distributed to several devices, executed on the devices and the result is sent back. All necessary distribution, coordination and synchronization is hidden beyond the systems interface. If a robot has to be moved in order to execute the actions, then also the entire movement process in hidden from the programmer and coordinated by the system.

Applications consist of systemic descriptions, more precisely a specification of actions and their corresponding constraints. The system provides a `schedule()` operation (system call) that requests system resources that are necessary for executing the application. For this the system schedules the actions in space and time. If an action becomes scheduled, a certain time slot of the executing device is reserved and, thus, the execution is guaranteed.

In many cases, there is a correlation between actions: actions depend on each other (sense-and-react) or a group of actions shall be executed at the same point in time and / or at certain locations. There are cases in which a non-executed action out of a group of actions has a significant impact on the overall outcome. Therefore, we introduce the transaction concept for spatial-temporal programming.

ActionSuites are containers for actions. All actions have to reside in exactly one suite. The suite is contained in the library and provides the transaction semantic for actions. Every suite is, therefore, a distributed transaction.

III. Transactions

Every SwarmAction is encapsulated into an ActionSuite. Depending actions must reside in the same suite, inter-suite communication is not possible. Usually, the failure of one action causes depending actions to be not executable and, therefore, the entire suite fails. Thus, a suite is a distributed transaction. We guarantee that if a suite is scheduled and finally, all participating nodes have commonly voted for *commit*, then the suite will be executed atomically. Hence, the application is aware in advance if a suite will fail or will execute successfully. Guarantees can only be given according to a given fault model. We assume here that no additional link or node failures will appear, i.e., there might be failures before the scheduling and

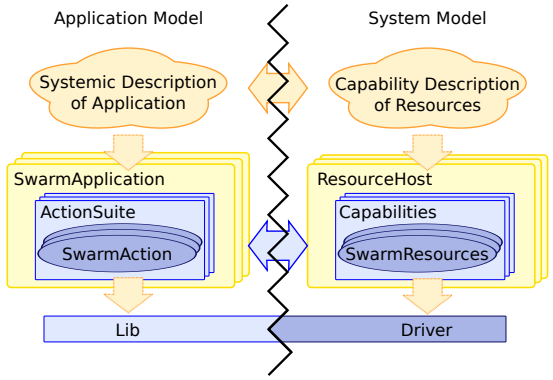


Fig. 1. Software Component Description

the voting, but, once the voting is done and the application is notified that the suite is committed for execution, we assume that no additional failures occur to the involved nodes of the distributed transaction.

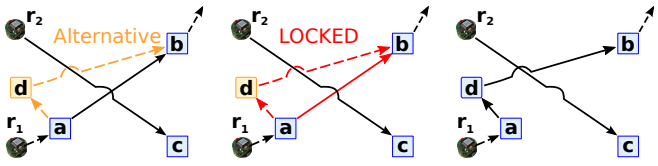
In order to guarantee the proper execution of an ActionSuite, we have to cope with *schedulability* and *executability*:

- **Schedulability**: The scheduler has to check general schedulability of the actions contained in the ActionSuite. So, the scheduler checks if all required resources can be made available under the given spatial-temporal constraints (this also includes physical movement). In case the scheduler successfully found a schedule, then the suite is called schedulable. Otherwise the suite is not schedulable.
- **Executability**: Schedulability is a necessary condition for executability. Executability is the state in which all nodes (which are involved in the distributed execution of the ActionSuite) jointly voted for commit and, thus, state that they are able to execute their assigned actions.

The system has global knowledge about the local schedule of each node. A local schedule consists of spatial-temporal jobs, i.e., actions that shall be executed and spatial-temporal trajectories that the node has to move along. If the system schedules a new suite, it assigns the actions to available nodes that are able to execute them and, if movement is necessary, it also computes and assigns spatial-temporal trajectories. Since we have a distributed system, the scheduled jobs have to be sent to the respective nodes which is done via message passing. Messages have to be sent over the network which requires time. Real-time communication is desirable, but cannot be guaranteed since we have wireless multihop communication. Therefore, there is no upper limit for the message delay. Since actions have timing constraints, the assignment of new jobs to a node has to be done in time. Late arrival of a message would in the worst-case result in not executing the actions.

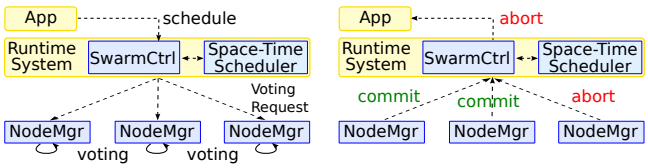
A. Path Alternatives

We define an uncertainty period between the point in time when the system computes a new schedule and the time the new schedule is accepted or rejected by the involved nodes. In order to cope with this uncertainty (and to avoid inconsistent



(a) Path Alternative (b) Lock Segment (c) Partial Commit

Fig. 2. The Problem of Time causes Path Alternatives



(a) Voting Phase (b) Commit Phase

Fig. 3. Time-Based Two-Phase Commit Protocol

knowledge between the system and the node), the computation of the new schedule is considered as an alternative to the current schedule. Figure 2(a) shows the following situation: There are three jobs in the system (a, b, c). Robot r_1's schedule contains a and b. The arrows indicate the movement direction. A new job d is scheduled by the system and assigned to r_1. The system assures schedulability, but not executability. Since this information has to be sent and acknowledged by r_1, the system is not able to drop the old schedule and, thus, creates an alternative route: $a \rightarrow d \rightarrow b$. Since the system performs a static collision avoidance and also a collision avoidance with mobile obstacles (other robots), the alternative has to be active as long as no decision is known. The alternative is then sent to r_1. If the message containing the alternative arrives too late, i.e., after the fork point at a, then r_1 continues moving on the old trajectory towards b. In this case, the new job d is implicitly rejected by the robot. If the message arrives before the robot reaches the fork point, the robot accepts the new trajectory. If, during the alternative path is valid, a new job e is requested for scheduling and r_1 is chosen again, then the path alternative has to be computed for every existing alternative.

Let us assume that e can only be scheduled before b, but after a and, if d is committed, then e has to be scheduled after d. Since d is neither committed nor rejected yet, the system has to compute the following path combinations: $a \rightarrow b$, $a \rightarrow e \rightarrow b$, $a \rightarrow d \rightarrow b$ and $a \rightarrow d \rightarrow e \rightarrow b$. The complexity of computing new path alternatives is exponential (2^x) in the number (x) of new jobs arriving in the system. In order to avoid this, the respective trajectory segment in which the alternative is computed is locked during the uncertainty period as depicted in Figure 2(b). The lock is kept until the respective node has decided to accept or reject the scheduled job d. Figure 2(c) shows the situation in which r_1 has decided to accept d and, thus, the old trajectory and the lock are removed. In case of a reject, the alternative is removed.

B. Time-Based Two-Phase Commit Protocol

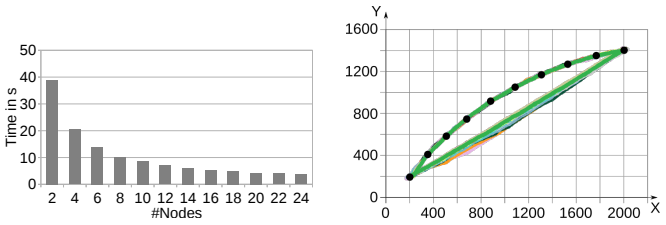
After the system has successfully checked schedulability, all nodes that are involved in the same transaction have to decide about executability. This is realized by a voting protocol which is based on the two-phase commit protocol (2PC). In the original 2PC, the period between issuing a local commit vote and waiting for the coordinator to either commit or abort is called uncertainty period. If the final message from the coordinator got lost, the node cannot unilateral commit the transaction because it does not know the global decision and it can also not unilateral abort because the coordinator may already have sent a global commit which got lost. In this case (considering a database system) the node keeps the respective tuple locked. If this is a permanent error then user interaction is required.

However, by only implementing the 2PC the mentioned case would be a severe problem. Assume a node moves to a fork point. Since the node is still uncertain it cannot go in the direction towards the new (not yet committed) job and it also cannot go in the other direction (which would indicate an abort) due to the same reason: it still does not know the global decision. Waiting at the fork point is also not an option since this would possibly violate all further (committed) jobs. Thus, the node is forced to move. Hence, we modify the original 2PC as depicted in Figure 3(a): After the system has computed a new schedule, the coordinator (*SwarmCtrl*) sends a voting request to all nodes participating in the transaction. Each node performs a local vote and sticks to its vote, i.e., locally commits/ aborts the schedule. Finally, it sends the result back to *SwarmCtrl* which leads to a partial commit/ abort to the schedule (Figure 3(b)). This results in immediately removing one of the path alternatives and releasing the lock. If at least one abort is received by the *SwarmCtrl* the transaction is globally aborted. If, however, partial commits are already executed, then *SwarmCtrl* triggers an `unschedule()` operation and tries to remove scheduled actions and reestablishes the old trajectory. We guarantee that if the application is notified of a global commit, then all nodes have commonly voted for commit and the actions in the transaction are committed for execution, i.e., resources are successfully reserved and, thus, executability is guaranteed.

IV. Proof-of-Concept

For performing experiments and showing a proof-of-concept, we have set up a testbed consisting of 40 mobile robots (400 MHz ARM9 CPU) and 24 stationary boards (180 MHz ARM9 CPU). Both devices are equipped with 64 MB SDRAM. The robots have two separately controllable electric engines. We use an external locating system for indoor tracking of the robots based on a Microsoft Kinect.

In order to show a proof-of-concept, we implemented two applications: a stationary distributed computation of the Mandelbrot set and a monitoring application along a trajectory that requires movement. The distributed computation of the Mandelbrot set is a suitable application since the resulting computational problem can be sub-partitioned into several chunks, each of which can be assigned to different nodes for execution. Each sub-partitioned problem was implemented as



(a) Mandelbrot Set (b) Observation

Fig. 4. Evaluation of Swarm Applications

an individual action. The execution policy here was to maximize the speed-up. We measured the total execution time of this application by creating up to 24 actions where each action was assigned to a separate robot. Figure 4(a) shows the degrading execution time of the distributed Mandelbrot set computation. The execution time with only two robots took 39 seconds while the execution time with 24 robots was around 3.5 seconds. Thus, this application scales well with the number of nodes.

The second application was a monitoring application along a programmer-defined trajectory of points. The application consists of 100 actions realized by iterating ten times over an ActionSuite of ten actions with spatial-temporal constraints. The timing constraints for each action were in ascending order such that a topological sorting on the actions is applicable. Considering the first suite with the first ten actions, the first action has a spatial constraint of (200, 200) and the tenth action has a spatial constraint of (2000, 1400). The eight remaining actions in-between those two actions are sorted in ascending order (concerning the spatial constraints). After we scheduled the first suite, we created a second suite with another 10 actions that have the same spatial constraints and increasing timing constraints. Figure 4(b) shows the traces of one robot that moved in total ten times along the trajectory. The visualized traces show the accuracy of the robot of keeping on an ideal trajectory.

V. Related Work

In [5], SwarmOS is presented as a mediation layer between applications and distributed resources. It has to cope with distribution, heterogeneous and shared resources as well as dynamic situations. In contrast to our approach, SwarmOS does not feature autonomous resource movement. The same holds for MagnetOS [1], which provides to split an application into components each of which will be dynamically assigned to several executing nodes (targeting ad-hoc and sensor networks). In [6], an approach for dynamic task assignment in robot swarms based on swarm algorithms with stochastic elements is presented. It supports groups of nodes that collectively execute applications without the need to program the nodes separately, i.e., it provides location transparency within the group.

The Symbrion and Replicator projects [3] consist of super-large-scale swarms of robots that are based on bio-inspired approaches featuring self-X properties. The systems are highly dynamic and so, if advantageous, the robots can aggregate into a symbiotic organism that is, in this form, better suited to solve a task while sharing resources such as energy. There are different kinds of programming abstractions for distributed, concurrent and parallel systems as well as for sensor networks. Detailed surveys are provided in [9], [7]. Following a holistic approach, nesC [2], which is an extension to C, is a programming language for deeply networked systems which was created for TinyOS. Programs are built from components that have internal concurrency. While nesC is a node-level language (code is written for an individual node), Pleiades [4] provides an abstraction to implement a central program that has access to the entire network (also known as macroprogramming [10]). SpatialViews [8] is an extension to Java which allows to define virtual networks that are mapped to physical nodes according to their physical location and the services they provide. Execution is distributed among the nodes in the virtual network performed by code migration.

VI. Conclusion

In this paper, we presented an approach to support transaction-based spatial-temporal programming of mobile robot swarms on a systemic level. The main reason for the modification of the 2PC protocol remains in the fact that at each time no inconsistencies between the global system knowledge and the local node knowledge may occur. In the worst-case an inconsistency could result in a robot crash since this could result in corrupted computation of spatial-temporal trajectories. According to our approach, we guarantee that this will not happen.

References

[1] R. Barr, J. C. Bicket, D. S. Dantas *et al.*, "On the Need for System-Level Support for Ad hoc and Sensor Networks," *Operating System Review*, vol. 36, pp. 1–5, 2002.

[2] D. Gay, P. Levis, R. von Behren *et al.*, "The nesC language: A holistic approach to networked embedded systems," *SIGPLAN Not.*, vol. 38, no. 5, pp. 1–11, May 2003.

[3] S. Kernbach, E. Meister, F. Schlachter *et al.*, "Symbiotic robot organisms: REPLICATOR and SYMBRION projects," in *Proceedings of the 8th Workshop on Performance Metrics for Intelligent Systems*, ser. PerMIS '08. New York, NY, USA: ACM, 2008, pp. 62–69.

[4] N. Kothari, R. Gummadi, T. Millstein *et al.*, "Reliable and efficient programming abstractions for wireless sensor networks," *SIGPLAN Not.*, vol. 42, no. 6, pp. 200–210, Jun. 2007.

[5] E. A. Lee, J. D. Kubiatowicz, J. M. Rabaey *et al.*, "The TerraSwarm Research Center (TSRC) (A White Paper)," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2012-207, Nov 2012.

[6] J. McLurkin and D. Yamins, "Dynamic task assignment in robot swarms," in *Robotics: Science and Systems Conference*, Cambridge, MA, USA, 2005.

[7] L. Mottola and G. P. Picco, "Programming Wireless Sensor Networks: Fundamental Concepts and State of the Art," *ACM Comput. Surv.*, vol. 43, no. 3, pp. 19:1–19:51, Apr. 2011. [Online]. Available: http://doi.acm.org/10.1145/1922649.1922656

[8] Y. Ni, U. Kremer, A. Stere *et al.*, "Programming ad-hoc networks of mobile and resource-constrained devices," *SIGPLAN Not.*, vol. 40, no. 6, pp. 249–260, Jun. 2005.

[9] R. Sugihara and R. K. Gupta, "Programming Models for Sensor Networks: A Survey," *ACM Trans. Sen. Netw.*, vol. 4, no. 2, pp. 8:1–8:29, Apr. 2008. [Online]. Available: http://doi.acm.org/10.1145/1340771.1340774

[10] M. Welsh and G. Mainland, "Programming sensor networks using abstract regions," in *Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation - Volume 1*, ser. NSDI'04. Berkeley, CA, USA: USENIX Association, 2004, pp. 3–3.