Context-Aware Dynamic Reconfiguration of Mobile Patient Monitoring Systems

Hailiang Mei, Bert-Jan van Beijnum, Pravin Pawar, Ing Widya, Hermie Hermens
Department of Electrical Engineering, Mathematics and Computer Science
University of Twente, The Netherlands.
{h.mei, Beijnum, p.pawar, i.widya, h.hermens}@cs.utwente.nl

Abstract

A mobile monitoring system for ambulant patients typically monitors bio-signals of a patient, processes and distributes these signals to formal caregivers. Often the monitoring service is long lasting, ranging from hours, days, to even months. The tasks involved in this end-to-end service are distributed over a networked ICT infrastructure and an optimal assignment may be found using task assignment techniques. To accommodate for the network and system resource dynamics, a configured system based on the optimal assignment over time may require a task redistribution. This capability is made possible using principles from autonomic computing where the operational environment is monitored, and based on significant operational changes a new optimal task assignment is computed, a reconfiguration plan is constructed and executed. In order for this approach to be effective, this control loop itself needs to be efficient, fast and minimally disruptive. In this paper we propose a computational scheme to estimate the quality of this control loop, the so-called reconfiguration cost. To this end, we propose an OSGi based task execution and control environment, a simple reconfiguration planning scheme based on the OSGi service bundle lifecycle, and a computational model to estimate the reconfiguration costs. Initial results using a prototype implementation are reported.[1]

1. Introduction

A mobile patient monitoring system [1] acquires, processes and transmits patients' bio-signals, e.g. ECG and blood pressure, to a clinical service point and dispatches a helping team to the patient's location in case of a medical emergency. This system needs to work 24 x 7 to provide a continuous monitoring service to the patient. However, similar to other applications operating in a mobile environment, the bio-signal processing and transmission performance of a patient monitoring system could be (deeply) affected by context changes and scarcity of resources, e.g. network bandwidth, battery power and computational power of handhelds. For example, a drop in network bandwidth due to patient's mobility can result in transmitted bio-signal loss or excessive delay. When this performance decrease exceeds certain tolerated level, the entire monitoring system may fail on responding accurately and timely to an emergency situation. Thus, the success of the mobile patient monitoring system relies heavily on whether the system can provide adequate and continuous bio-signal processing and transmission services despite the context variations. As shown in Figure 1, a typical model of patient monitoring system consists of a set of bio-signal processing and transmission tasks (a monitoring application) deployed across a set of distributed resources (a device network). In the previous work [2], we have shown that an optimal task assignment may be found using task assignment techniques. To accommodate network and system resource dynamics, a configured system based on the previous optimal assignment can be dynamically reconfigured according to new optimal assignment computed under the new situation.

In the dynamic reconfiguration of mobile patient monitoring system, the execution states are often required to be preserved in the target configuration since they might contain important medical information derived from earlier received inputs. Thus, we focus on the reconfiguration supporting states transfer, i.e. stateful reconfiguration. In this paper, based on a task distribution framework implemented using OSGi technology[2] for mobile patient monitoring systems, we propose a stepwise scheme to plan the stateful reconfigurations.

Furthermore, we present a computational model to estimate the reconfiguration cost in terms of reconfiguration time. The benefit of this model is twofold. (1) The stateful reconfiguration could cause a potential disruption at the bio-signal processing and transmission services. It is rather crucial to estimate the scale of this disruption before executing the reconfiguration plan. (2) When a reconfiguration decision is required given a certain context change, there could exist

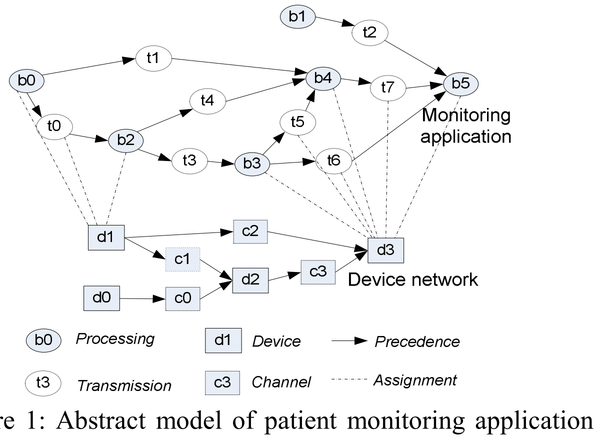

Figure 1: Abstract model of patient monitoring application and device network

[1] This work is supported by Dutch Freeband Awareness project (under grant BSIK5902390, http://awareness.freeband.nl/)

[2] www.osgi.org

multiple new task assignments that offer similar performance compared to the optimal one under the new situation. To choose the best, planners need to balance the tradeoff between the performance of these new task assignments and their reconfiguration complexity.

2. Related work

Dynamic reconfiguration is a technique to improve service availability since it can update a running system without taking it off-line [3]. This technique has been applied to deal with changing user requirements [4] and underlying resources [5] in distributed streaming applications, which is closely related to mobile monitoring system. Dynamic reconfiguration also resembles the concept of code mobility [6]. In this paper, we tackle a strong mobility problem where both the code and the execution state are migrated to a different node. The development on component-based software engineering have enabled efficient implementation of dynamic reconfiguration system, e.g. CORBA [7], enterprise JavaBean [8]. In this direction, we base our design and implementation on OSGi technology that fits resource constrained devices better. The research on reconfiguration cost has been lacking. In [9], it is quantified as proportional to the number of affected reconfigurable entities. In this paper, we take a step further to quantitatively measure the reconfiguration cost in a heterogeneous environment.

3. Bio-Signal Processing Unit distribution framework

In this section, we present a task distribution framework as part of the patient monitoring system that supports monitoring, planning and execution for dynamic reconfiguration. We select OSGi technology as the base for our design and implementation because of its advantage on standardization and compactness.

Table 1: Two different task assignments "α" and "β" for the system shown in Figure 1. The affected tasks are highlighted.

	α (original)	β (target)
b0	d1	d1
b1	d0	d0
b2	d1	d3
b3	d3	d3
b4	d3	d3
b5	d3	d3
t0	d1	d1-c2-d3
t1	d1-c2-d3	d1-c2-d3
t2	d0-c0-d2-c3-d3	d0-c0-d2-c3-d3
t3	d1-c1-d2-c3-d3	d3
t4	d1-c1-d2-c3-d3	d3
t5	d3	d3
t6	d3	d3
t7	d3	d3

3.1. BSPU – a unit of system composition

In mobile patient monitoring system, a particular system configuration can be represented by a unique assignment function from a set of tasks onto a set of resources such that: each processing task is assigned to one of the available devices and each transmission task is assigned to one of the available communication paths. A communication path is a directed path connecting two devices in a device network. For example, in Figure 1, both "d1-c1-d2-c3-d3" and "d1-c2-d3" are communication paths connecting "d1" to "d3". Note that the communication path could also be a stream pipe within the device and is denoted the same as the device "di". Two different task assignments ("α" and "β") for the system modeled in Figure 1 are illustrated in Table 1.

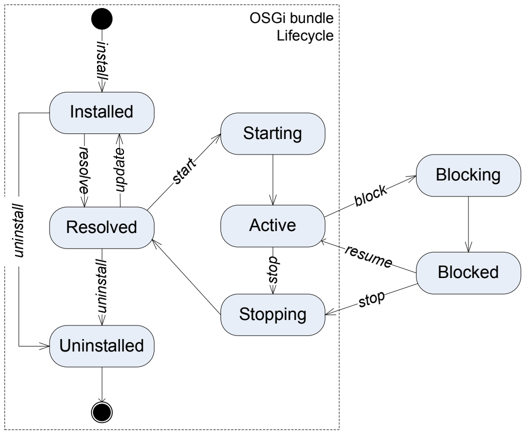

Figure 2: Lifecycle of BSPU

To support the execution of mobile monitoring application, we design a software component named as BSPU (Bio-Signal Processing Unit) which plays as a unit of system composition. A BSPU receives continuous stream of bio-signals via its input buffer from its predecessor BSPUs, manipulates the stream depending on its functionality if necessary, and streams the processed results to the successor BSPUs if any. We can distinguish two types of BSPU: *processing* and *relaying*. Each *processing* BSPU implements a processing task, e.g. "b2" in Figure 1 can be viewed as a processing BSPU[3]. Relaying BSPUs do not contain any data manipulation functions and are implemented to support the execution of transmission task. We denote the relaying BSPU of a transmission task "ti" as "ti*". For example, one "t3*" should be deployed onto device "d2" to support the execution of "t3" in the task assignment "α".

A BSPU provides a set of control interfaces so that an external entity can manage its lifecycle. With the choice of OSGi technology, the BSPU lifecycle model is designed by extending the OSGi bundle lifecycle, i.e. adding two new states and two new transitions (Figure 2). The introduction of new

[3] In this paper, we use the same notation of "bi" to represent a processing task or a processing BSPU. When it occurs, the exact meaning should be clear from the context.

states and transitions can be used to temporarily freeze a set of BSPUs and allow stateful reconfiguration. A "block" method enforces a transition to "Blocking" state. In this state the BSPU will no longer take incoming data streams from its input buffer but just finish the ongoing processes. After it finishes all the ongoing processes, this BSPU will move to "Blocked" state where it no longer sends any data streams to its successor BSPUs. The last message sent from a BSPU before it moves into "Blocked" state contains a "Blocked" flag. This flag will be propagated through all successor BSPUs to inform them about the applied "block" action. A "resume" call unblocks a "Blocked" BSPU and enables it to process again the incoming data streams in its input buffer. When a BSPU is in the state "Active" or "Blocked", it is possible to fetch/set the execution state information through its control interface.

3.2. Architecture

The important entities in the BSPU distribution framework are a Coordinator and a set of Facilitators (Figure 3). A Facilitator is the representative of its device in the distribution framework: it reports the presence/absence of the device, it receives control commands on BSPU life-cycle management from the Coordinator and reports errors if any, back to the Coordinator. Once a stateful reconfiguration is initiated by the Coordinator, the execution state information of BSPU is examined by its hosting Facilitator and transferred (via Coordinator) to the new BSPU.

The Coordinator runs a task assignment algorithm [2] that computes the optimal task assignment based on the required monitoring application and the current device network's context information, e.g. available devices and their connectivity, device's CPU load, device's remaining battery energy, etc. Once a significant change occurs in the required monitoring application or in the device network's context, the Coordinator is triggered to compute, under the new situation, the optimal task assignment together with a few near-optimal assignments as candidates. These candidate assignments are further ranked subject to both their performance enhancement compared to the current assignment and their reconfiguration cost (c.f. Section 4). If the identified best assignment is different from the current one deployed in the system, the Coordinator translates this difference in task assignments into a reconfiguration plan (c.f. Section 3.4) over a set of *affected BSPUs* (c.f. Section 3.3). Then, the Coordinator controls the hosting Facilitators of affected BSPUs to deploy the new task assignment by means of BSPU redistribution.

3.3. Affected tasks and affected BSPUs

An original task assignment and a target task assignment are different if at least one task is subject to location change, i.e. addition, relocation and removal. In a set of connected tasks, a location change to one task will require certain adjustments to other tasks that depend on this task's behavior. In monitoring application, the dependency between two tasks is in line with their precedence relation. For example, as shown in Figure 2, the behavior of "b2" depends on "t3" and "t4" since "b2" requires the appropriate operations and settings of "t3" and "t4" to transfer its output data stream. "t3" further depends on "b3" since "t3" requires the appropriate location and settings of "b3" to process its transferred data stream. For example, in the reconfiguration from "α" to "β" (Table 1), we should not only consider the tasks whose locations are explicitly changed, i.e. "b2", "t0", "t3" and "t4", but also those tasks that are affected implicitly, i.e. "b0".

We name all these tasks subject to reconfiguration as *affected tasks*:

- A transmission task is affected if it is added, relocated or removed.
- A processing task is affected if (1) it is added, relocated or

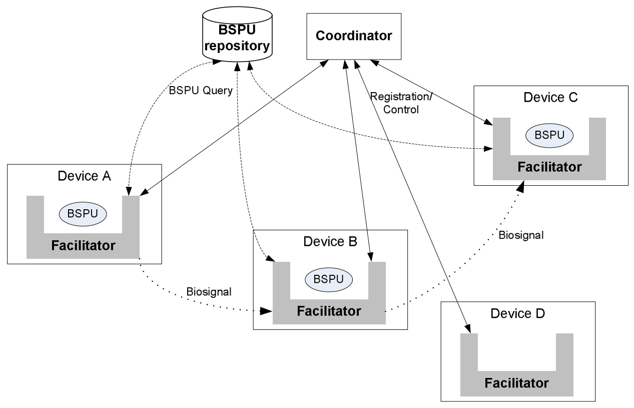

Figure 3: BSPU distribution framework

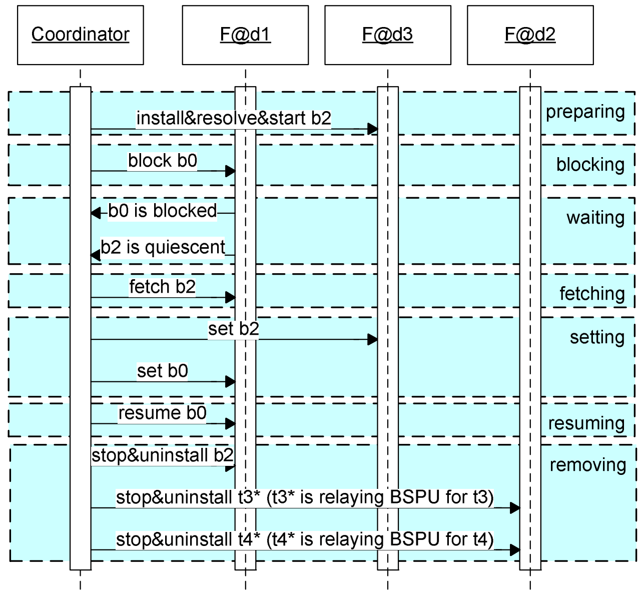

Figure 4: The seven-step reconfiguration plan applied to support the reconfiguration from "α" to "β"

removed or (2) any of its outgoing transmission tasks is affected.

Thus we can identify, for any reconfiguration, the set of affected tasks unambiguously as shown in Table 1. In the BSPU distribution framework, a reconfiguration plan only addresses those processing BSPUs that implement affected processing tasks and those relaying BSPUs that support affected transmission tasks.

3.4. Reconfiguration plan

In order to maintain the state consistency, a three-stage reconfiguration approach is proposed in [7]: (1) drive system to a safe-state in which those to-be-reconfigured components are self-contained and stable and none of them is involved in the interactions; (2) detect if the safe state has been reached; and (3) apply reconfigurations.

Following this general approach, we define a sequential seven-step scheme to create dynamic reconfiguration plan. Note that the reconfiguration should be treated as an atomic transaction: in case any one action fails during the execution, the actions taken so far must be rolled back as to retain the original configuration. These seven steps are explained as follows with the example of reconfiguration plan from "α" to "β" as depicted in Figure 4:

1. *Preparing*: Before interrupting the on-going patient monitoring application, Coordinator installs and starts every relocated and newly added BSPU at the targeted new location, e.g. "install&resolve&start b2". Thus, in case any action fails in this step, there is no penalty on the on-going monitoring application.
2. *Blocking*: In this step, the on-going monitoring application is interrupted. Coordinator identifies and blocks all the so-called *source affected BSPUs* in the original configuration, e.g. "block b0". A *source affected BSPU* is an affected BSPU none of whose predecessor BSPUs are affected.
3. *Waiting*: after a source affected BSPU enters "Blocked" state, it sends a notification via its hosting Facilitator to Coordinator, e.g. "b0 is blocked". After a BSPU receives a "Blocked" flag from its precededecessor BSPU and finishes all possible processing of the remaining data stream in its input buffer, it is still in "Active" state but will not perform any operations until the incoming data stream fully resumes. This BSPU should notify Coordinator via its hosting Facilitator about this quiescent situation, e.g. "b2 is quiescent". After all the notifications from affected BSPUs are received by the Coordinator, the system enters the safe state for reconfiguration.
4. *Fetching*: Coordinator fetches the current execution state information from every relocated BSPU in the original configuration, e.g. "fetch b2"
5. *Setting*: Coordinator sets the execution state at every relocated BSPU at the targeted new location, e.g. "set b2". Coordinator sets the BSPU whose configuration parameters are required to update, e.g. "set b0".
6. *Resuming*: Coordinator resumes all source affected BSPUs, e.g. "resume b0". After this step, the patient monitoring system should resume to work normally in the target configuration.
7. *Removing*: Coordinator removes all the outdated BSPUs (including relaying BSPUs) left from the original configuration, e.g. "stop&uninstall b2" and "stop&uninstall t3* and t4*".

Overall, the execution of the reconfiguration plan can be viewed as a combination of a set of *control communications* (including both request and reply) between Coordinator and Facilitators and a set of *control actions* on BSPUs applied by their hosting Facilitators.

4. Reconfiguration cost

From the description in the Section 3, we can see that the time duration of a reconfiguration plan execution is closely related to the difference between the original task assignment and the target task assignment: the larger the difference, the more BSPUs are affected and thus the more time spent on control communications and actions. We define the following cost functions to estimate the time duration of each control communication and control action observed at the Coordinator:

- c^x(*Facilitator*) denotes the time duration of a control communication between the Coordinator and a particular Facilitator. Due to the communication heterogeneity, this time duration is a function of the communication channel between Coordinator and Facilitator. For simplicity, we define this cost as a function of Facilitators.
- c^a(*Facilitator, BSPU, Action*) denotes the time duration of a control action. Due to the device heterogeneity and different complexity level of actions, this time duration is a function of the set of Facilitators (each representing its hosting device), the set of BSPUs, and a set of actions, where *Action* = {block, resume, install, uninstall, resolve, start, stop, fetch, set, wait}

The Coordinator's knowledge about these two cost functions can be learnt through the system operation. Initially, the Coordinator simply associates a set of heuristic values as the function results. From time to time, the Coordinator can update these values based on the measurements of real communications or actions and thus to tune the cost functions to provide more accurate results. In the Table 2, we present the reconfiguration cost of each individual step of the reconfiguration plan based on the proposed cost functions and the reconfiguration example shown in Figure 4. In the proposed reconfiguration plan, only from step 2 (blocking) to step 6 (resuming), the monitoring application potentially suffers from service disruption. Thus, the total reconfiguration cost of a particular reconfiguration in our case is the combination of the costs of these five steps.

5. Experiment results

We implement the BSPU distribution framework based on the OSGi technology with an OSGi added-on service, i.e. R-OSGi[4] that offers the remote connection support between distributed OSGi frameworks. BSPUs, the Facilitator, and the Coordinator are all implemented as OSGi bundles. To validate the sequential seven-step dynamic reconfiguration scheme, in particular to evaluate the reconfiguration cost, we experiment the dynamic reconfiguration (Figure 4) from task assignment "α" to "β" in two settings. In setting #1, the Coordinator and all Facilitators are located on one Windows XP machine. In setting #2, the Coordinator and F@d1 are located on one XP machine while F@d2 and F@d3 are each located on a different XP machine in the same LAN. All the functionalities of the framework are supported except for the handling of "blocked" and "quiescent" messages. We run dynamic reconfiguration experiment on each setting 5 times. We report the average time duration of each reconfiguration steps in Figure 5. We can see that the time duration of "preparing", "setting" and "removing" steps in setting #2 are significantly higher than those in setting #1. This difference is caused by the fact that all control communications in setting #1 are done within the same machine, whereas in setting #2, some control communication are over the LAN.

Table 2: Stepwise reconfiguration costs from "α" to "β"

Step	*Cost*
preparing	$c^x(F@d3) + c^a(F@d3, b2, install) + c^a(F@d3, b2, resolve) + c^a(F@d3, b2, start)$
blocking	$c^x(F@d1) + c^a(F@d1, b0, block)$
waiting	$c^a(F@d1, b0, wait) + c^a(F@d1, b2, wait)$
fetching	$c^x(F@d1) + c^a(F@d1, b2, fetch)$
setting	$c^x(F@d3) + c^a(F@d3, b2, set) + c^x(F@d1) + c^a(F@d1, b0, set)$
resuming	$c^x(F@d1) + c^a(F@d1, b0, resume)$
removing	$c^x(F@d1) + c^a(F@d1, b2, stop) + c^a(F@d1, b2, uninstall) + c^x(F@d2) + c^a(F@d2, t3^*, stop) + c^a(F@d2, t3^*, uninstall) + c^x(F@d2) + c^a(F@d2, t4^*, stop) + c^a(F@d2, t4^*, uninstall)$

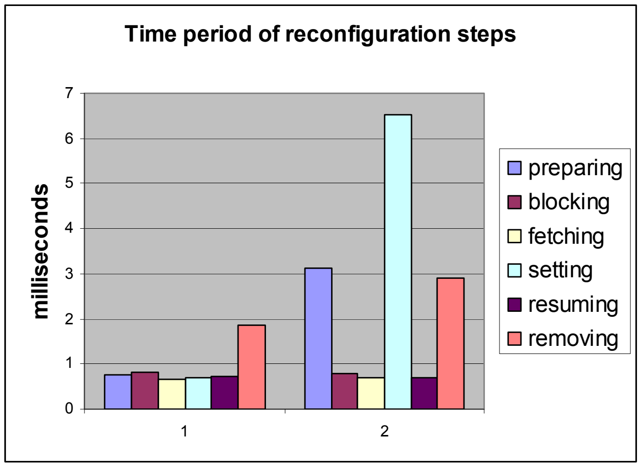

Figure 5: Dynamic reconfiguration experiments in two settings

6. Conclusion

In this paper, we propose an OSGi based task distribution framework to support context-aware dynamic reconfiguration in mobile patient monitoring system. In particular, a reconfiguration planning scheme based on the OSGi service bundle lifecycle and a computational model to estimate the reconfiguration costs are presented. Initial results using a prototype implementation are reported. Our future works include more extensive experiments in heterogonous networks and more detailed analysis on reconfiguration costs.

References

1. Jurik, A.D. and A.C. Weaver, Remote Medical Monitoring. Computer, 2008. 41(4): p. 96-99.
2. Mei, H., Smart Distribution of Bio-Signal Processing Tasks in M-health, in 4th OTM Academy Doctoral Consortium. 2007: Vilamoura, Algarve, Portugal.
3. Wegdam, M., PhD Thesis: Dynamic Reconfiguration and Load Distribution in Component Middleware. 2003, University Twente, The Netherlands.
4. Gu, X. and K. Nahrstedt, On Composing Stream Applications in Peer-to-Peer Environments. Parallel and Distributed Systems, IEEE Transactions on, 2006. 17(8): p. 824-837.
5. Amini, L., et al. Adaptive Control of Extreme-scale Stream Processing Systems. in 26th IEEE International Conference on Distributed Computing Systems (ICDCS'06). 2006.
6. Fuggetta, A., G.P. Picco, and G. Vigna, Understanding Code Mobility. IEEE Transaction on Software Engineering, 1998. 24(5): p. 342-361.
7. Almeida, J.P.A., et al. Transparent dynamic reconfiguration for CORBA. in 3rd International Symposium on Distributed Objects and Applications (DOA '01). 2001.
8. Rutherford, M.J., et al. Reconfiguration in the Enterprise JavaBean component model. in IFIP/ACM Working Conference on Component Deployment. 2002.
9. Moazami-Goudarzi, K., Consistency preserving dynamic reconfiguration of distributed systems, in Imperial College London. 1999.

[4] http://r-osgi.sourceforge.net/