Task Distribution Based Adaptation in Mobile Patient Monitoring Systems

Hailiang Mei

Graduation committee:

Chairman:	Prof. dr. ir. Anton J. Mouthaan
Promotor:	Prof. dr. ir. Hermie Hermens
Promotor:	Prof. dr. ir. Boudewijn R. Haverkort
Assistant promotor:	Dr. ir. Bert-Jan van Beijnum

Members:

Prof. dr. Wouter Joosen	Katholieke Universiteit Leuven
Prof. dr. Johann Hurink	University of Twente
Prof. dr. ir. Bart Nieuwenhuis	University of Twente
Prof. dr. Miriam Vollenbroek-Hutten	University of Twente
Dr. ir. Marten van Sinderen	University of Twente

CTIT Ph.D.-thesis Series No. 10-173
Centre for Telematics and Information Technology
University of Twente, P.O. Box 217, NL-7500 AE Enschede

ISSN 1381-3617
ISBN 978-90-365-3026-2

Publisher: Wöhrmann Print Service.

TASK DISTRIBUTION BASED ADAPTATION IN MOBILE PATIENT MONITORING SYSTEMS

PROEFSCHRIFT

ter verkrijging van
de graad van doctor aan de Universiteit Twente,
op gezag van de rector magnificus,
prof. dr. H. Brinksma,
volgens besluit van het College voor Promoties,
in het openbaar te verdedigen
op woensdag 2 juni 2010 om 16.45 uur

door

Hailiang Mei

geboren op 6 maart 1978
te Beijing, China

Dit proefschrift is goedgekeurd door:
Prof. dr. ir. Hermie Hermens (promotor)
Prof. dr. ir. Boudewijn R. Haverkort (promotor)
Dr. ir. Bert-Jan van Beijnum (assistent-promotor)

Abstract

In the past decades, telemedicine applications have been widely recognized as a key technology to solve many problems faced by our aging society. Being part of this technological move, Mobile Patient Monitoring Systems (MPMSs) have attracted a great deal of attention due to its fitness on providing less intrusive and long-lasting telemedicine services. However, similar to other applications operating in a mobile environment, the performance of an MPMS is affected by context changes and scarcity of resources, e.g., network bandwidth, battery power, and computational power of handhelds. To address such a demand and supply mismatch problem, we propose an adaptation mechanism for MPMSs that can adjust the assignment of tasks across available devices at run-time. The rationale is that, at any point in time, if one device cannot support a task for its computation or data communication demands, some other devices with richer resources might be able to take over this task. The advantage over other methods is that the user requirements are less likely to be compromised and that distributed resources are better utilized. This thesis covers two major research topics: the computation of a suitable task assignment and the dynamic distribution of tasks across devices according to this new assignment at run-time.

The first main contribution of this thesis is a set of task assignment algorithms designed for MPMSs. We propose two graph-based polynomial-time algorithms, which deal with the "chain-chain assignment" and the "tree-star assignment". When the targeted MPMS complies to such models, we recommend the use of these graph-based algorithms since they provide an optimal solution and their computation times are bounded. For MPMSs with a more generic topology, we propose an A*-based task assignment algorithms that can find the optimal solutions. Furthermore, a bounded approach is introduced for using the A*-based algorithms, which offer near-optimal solutions, yet can finish in a bounded time. Another main contribu-

tion is the study of a task distribution infrastructure for MPMSs. To enable system dynamic reconfigurations and to hide the heterogeneous nature of MPMSs, a middleware level solution named MADE (Monitoring, Analysis, Decision, and Enforcement) is proposed in this thesis. In particular, we model and examine the level of service interruption caused by the dynamic reconfiguration in MPMSs. Not only does this kind of reconfiguration cost influence the decision on whether a system reconfiguration should be executed or not, but it is also a crucial input for identifying a "real suitable" task assignment for an MPMS to adapt.

Acknowledgement

This thesis would not have been possible without the guidance, encouragement, and help from many colleagues, friends, and my family. I like to take this opportunity to express my sincere appreciation to all of them.

First of all, I start by thanking my promotors Prof. Hermie Hermens and Prof. Boudewijn Haverkort for their guidance on my PhD thesis. Their detailed comments and timely responses made the writing of this thesis smooth and pleasant. Hermie, thank you for providing me high quality suggestions and insights on the telemedicine applications. They helped me to target at the right problems. Boudewijn, thanks for suggesting interesting models and techniques and the constructive feedbacks on my thesis. I very much appreciate your encouragement on investigating deeper into the formulated problems. I also would like to give my special thanks to my daily supervisors, Dr. Bert-Jan van Beijnum and Dr. Ing Widya. They have guided me through the entire PhD trajectory and played a crucial role on setting up my research topic. I enjoyed the freedom they gave me on pursuing my research ideas. Their continuous questions have ensured me not to deviate too far away from an effective research path. Bert-Jan, thank you for providing practical suggestions on the algorithm designs and implementations. I also appreciate all the enjoyable chats we had that are beyond the research itself. Ing, thank you for the generous hours you spent with me on the in-depth discussions on problem formulation and modeling. I have also learnt so much from you on coaching students.

I am honored to have Prof. Johann Hurink, Prof. Wouter Joosen, Prof. Bart Nieuwenhuis, Prof. Miriam Vollenbroek-Hutten, and Dr. Marten van Sinderen in my graduation committee. I thank them for accepting the invitation and reviewing my thesis.

My PhD research is part of the Freeband Awareness project, I like to thank all the project partners for their kind support and fruitful discussions. Further, I extend my

gratitude to my colleagues in the Telemedicine/BSS group and the former ASNA group: Val, Richard, Pravin, Jan-Willem, Wies, Thijs, Peter, Vishwanath, Tom, Luiz Olavo, Teduh, Laura, Eduardo, Ricardo, Marten, Luis, Dick, Maarten, Tiago, Patricia, Joao Paulo, Rodrigo, Boris, Aart, Kamran, Annelies, and many others. I have spent some wonderful years working with all of you and appreciate the cheerful working environment that you help to create. A few months ago, I started working with some great new colleagues in the Netherlands BioInformatics Centre. Rob, Bharat, Kees, Christine, Barend, thank you very much for being supportive and considerate during my career transition.

Besides inspiring colleagues, I always feel very fortunate to have many great friends: Lian and Lily, Tao and Yu, Wenlong and Nong, Defeng and Xue, Yan and Dongni, Cynthia and Albert, Dapeng, Yanbo, Zhou, Cindy, Duanyang, Jia Wang, Daniela, Jiajia and Jianbin, Hans and Rinske, Bas, Eric and Lia, Gerard, Gijs, Yongfeng and Qun, Hong Tong, Jia Tian, Stefan, Peter, Maggie, Carmen, Nadine, Fei, Xiao. Thank you all for always being there to help me during the difficulties and share the joys and other great moments with me. Thanks also go to Jan, Djoerd and other soccer players in the UT-Kring club.

Being thousands miles away from my parents makes the feeling of connecting to them even stronger. Mum and Dad, you deserve my highest respect and deepest gratitude for your constant support and encouragement. Haiqing, thank you for all your suggestions on my career development and taking care our parents while I am restricted by the geographical distance. Last but not least, I would like to express my great gratitude to my beloved wife. Ting, simple words can not tell how much I feel blessed to meet you and marry you. Life has become so much easier and happier. We have walked through our PhD journeys together. How wonderful is that! Love and joy will fulfill the road ahead of us and thank you for being there with me.

Hailiang Mei
Rotterdam, May 2010

Contents

Chapter 1
Introduction

Mobile Patient Monitoring Systems (MPMSs) are being developed to provide high quality healthcare services in the near future. However, the gap between the application demands and resources still remains and may hinder this process. This PhD thesis proposes an adaptation mechanism for MPMS based on a dynamic task distribution approach. Through this adaptivity, an MPMS can dynamically alter its system configuration when necessary. The ultimate goal of this adaptation mechanism is to guarantee a certain quality level for delivered healthcare services despite varying demand and resource fluctuations. This chapter is organized as follows. Section 1.1 introduces the research background. Section 1.2 analyzes current MPMSs and motivates our work. Section 1.3 elaborates a set of research questions and the research approach. Section 1.4 presents the structure of this thesis.

1.1 Research Background

This section presents the research background of this thesis. First, we explain the concepts of telemedicine and mobile healthcare. Second, we introduce one example of medical services that can be provided by MPMSs, i.e., the epileptic seizure detection.

1.1.1 Telemedicine and Mobile Healthcare

In the past decades, telemedicine applications have been widely recognized and argued as a key technology to address many healthcare problems faced by our society. In [Rei96], *telemedicine* is defined as "the use of telecommunication technologies to provide healthcare services across geographic, temporal, social, and cultural barriers". According to the findings in [All04], the key drivers behind the move towards telemedicine can be summarized as follows:

- *An ageing population*: In Europe, by 2050, 4 in 10 people will be over 65 years old and many of them will require extensive healthcare services [All04]. On the other hand, the lack of human and financial resources will only become

more prominent beyond today's already stretched budgets. The practice of telemedicine can help to fill this resource gap by improving the efficiency of healthcare delivery.

- *Quality of care and care delivery*: Home care or "hospital at home" is the key element for improving the quality of care for patients of all ages. This requires that the patient's medical condition being continuously monitored and abnormalities being reported to the associated healthcare professional. Telemedicine systems provide the essential connection between patients and healthcare professionals.

- *Cost*: Medical treatments are getting more expensive and the national healthcare systems in many countries become unsustainable. Therefore, it is of great interests for governments and healthcare institutions to apply advanced Information and Communication Technology (ICT) to bring down the cost of delivering healthcare services.

- *Technology*: Technology, e.g., Internet and computers, has become cheaper and easier to use. This permits the adoption of ICT based healthcare delivery (including telemedicine) in everybody's daily life.

Within the domain of telemedicine, the study on mobile healthcare technology has been established as a new interdisciplinary area [IJZ04, ILP05]. In particular, various MPMSs [JW08] are emerging along with the widespread adoption of advanced sensor and mobile technology. An MPMS acquires patient's bio-signal information using bio-sensors, processes these data based on medical signal processing algorithms, distributes to a formal caregiver or a clinical decision support system, and stores them appropriately for later use or evaluation. Main users of an MPMS include patient, formal caregiver or other interested parties, e.g., relatives of the patient. We define the underlying computation and communication resource of MPMS as an *m-health platform*. A typical example of m-health platforms consists of multiple configurable sensors, a handheld device, a local server, a back-end server, an end-terminal, and the communication infrastructure connecting them (Figure 1.1). On top of the m-health platform, various telemonitoring applications can operate continuously (24×7 hours per week). Thus, an MPMS can be viewed as a combination of the underlying m-health platform and several deployed telemonitoring applications. Examples of telemonitoring applications include safety-critical applications such as trauma care, detection of life threatening ventricular arrhythmias, detection of foetal distress and premature labour [JHW$^+$06], and detection of epileptic seizures [THVH06]. Epileptic seizure detection is refereed through out this thesis

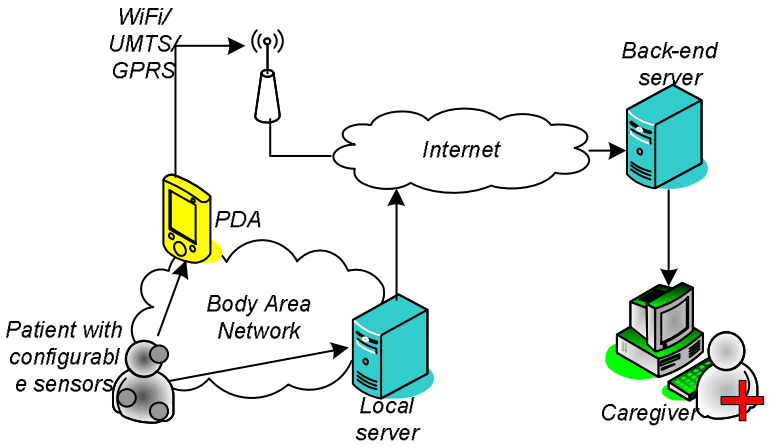

Figure 1.1: *An m-health platform*

as the main example of telemonitoring applications, thus we will explain it in more detail in the next section.

1.1.2 Epileptic Seizure Detection

Epilepsy is a serious chronic neurological condition characterized by recurrent unprovoked seizures. Early detection and prediction of seizures, even for just a few seconds, could give patients a chance to prepare and receive appropriate medical assistance or advice. One existing seizure detection/prediction algorithm [THVH06] is depicted in Figure 1.2. It operates as follows. First, the patient's raw Electro-Cardio Gram (ECG) signal is filtered to remove signal artifacts and environment noise. Second, heart beats are detected in the "R-top detection" step and then Heart Rate Increase (HRI) is calculated. A task ("HR event detection") then detects the specific event in heart rate changes that associates with the symptom prior to an epileptic seizure attack. To reduce the chance of false alarms, the patient's activity and postural information are also measured and correlated with his heart beat information. Epileptic seizures may happen anywhere and at any time. This requires the detection algorithm to work 24×7 for providing a continuous monitoring service to epilepsy patients. An MPMS is a well suited candidate to offer the long-lasting computation and communication services. A scenario of seizure detection is illustrated as follows [vSBM05].

Scenario, part 1: John is an epileptic patient who has been seizure-free for several years. He wears an MPMS that monitors his health state and can give him a few seconds' advance

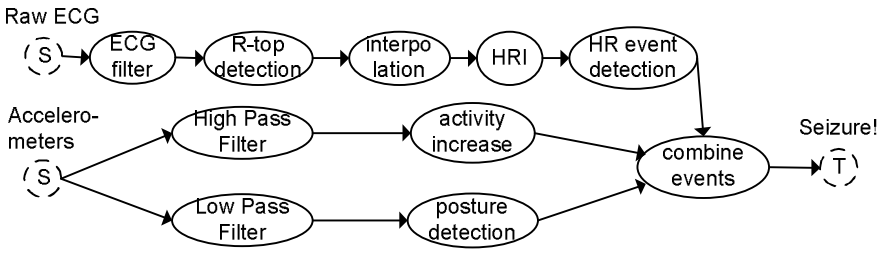

Figure 1.2: *A seizure detection algorithm*

warning of an upcoming seizure. When John is at home, a broadband network is available to transfer his raw ECG and activity information to the remote monitoring center, e.g., the back-end server in Figure 1.1. In this case, all tasks of the detection algorithm are deployed on the back-end server and John's raw bio-signals are stored safely. His doctor is warned if a seizure is likely to occur. Thus John feels very safe because he knows he can get immediate help once he has a seizure attack.

1.2 Problem Analysis

Similar to other applications operating in a mobile environment, an MPMS 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 performance drops below a certain level, the entire MPMS may fail in responding accurately and timely to an emergency [JIT+06]. Thus, the success of an MPMS relies heavily on whether the system can provide adequate and continuous bio-signal processing and transmission services despite context variations.

From a technological point of view [Sat04], when a mismatch occurs between a task resource demand and the system resource supply, there are three approaches for implementing an adaptation mechanism to tackle it:

1. *To inform the user*: This is to suggest a corrective action to the user.
2. *To reserve resources*: This is to ask the environment to guarantee a certain level of the availability of a resource, e.g., QoS management and reservation techniques.
3. *To adjust the application*: This is to automatically change application task behavior to use less of a scarce resource, e.g., scalable video transmission over

wireless network.

The first approach tries to avoid the mismatch by giving users a suggestion or warning. For example, in an MPMS, we could ask a patient using a telemonitoring application to stay near to a charging point to reduce the risk caused by draining battery power. However, restricting user mobility in this way is far from a satisfactory solution. The second approach assumes that it is possible to reserve sufficient resources for performing a task. This is sometimes unrealistic, e.g., the drop of network bandwidth could be so significant that the required data transmission rate just cannot be met.

In this thesis, we follow the third approach, which is to adjust task behavior to tackle the mismatch. Previously, adjusting application was often performed within an isolated device, e.g., by a local application-specific adaptor [BFK+00]. Methods applied in the past include data compression, discarding less important information, and handover to a better network connection. A unique feature of MPMSs is a distributed processing paradigm in which a set of processing tasks, e.g., the seizure detection algorithm (Figure 1.2), is spread across a heterogeneous network. Therefore, one possible adaptation mechanism is to explore this distributed processing paradigm and adjust the assignment of tasks across available devices at run-time. We refer to this mechanism as *task-distribution-based adaptation mechanism*. The rationale is that, at a particular point in time, if one device cannot support a task for its computation or data communication demands, some other devices with richer resources can take over this task. The advantage over traditional methods is that the user requirements are less likely to be compromised and distributed resources can be better utilized. The following scenario illustrates how an MPMS can adapt to changing contextual factors by dynamic task distribution.

Scenario, part 2: One afternoon, John is out jogging, following his usual route through the forest. Since there is no broadband network available in the forest, John's bio-signals cannot be transmitted due to insufficient network bandwidth. Therefore, some processing tasks are reassigned from the remote server to his Personal Digital Assistant (PDA) so that his bio-signals are processed locally. During his run, the signal processing algorithm detects a possible imminent epileptic seizure. John is immediately warned by his Body Area Network (BAN) and stops running. At the same time, an alarm and John's Global Positioning System (GPS) position are sent to the monitoring center via a narrow band connection, e.g., General Packet Radio Service (GPRS) or Global System for Mobile communications (GSM). The alarm triggers the monitoring service to take appropriate action. For example, depending on the circumstances, a medical team or an informal caregiver can be dispatched to the exact location where John is to render emergency assistance.

1.3 Research Questions

The task-distribution-based adaptation mechanism for MPMSs works as an automatic control loop that consists of the following three steps: (1) observe a mismatch in the present system configuration of an MPMS, (2) derive a new suitable task assignment where tasks can be supported by the assigned resources, and (3) reconfigure the MPMS accordingly. The need for better understanding this adaptation mechanism leads to the following five research questions:

- RQ1: What do we require for a "suitable task assignment" in an MPMS?

For an MPMS, i.e., an m-health platform and deployed telemonitoring applications, there are a number of possible system configurations resulting from different task assignments. Each system configuration exhibits different performance characteristics. In order to identify a suitable task assignment, we need to evaluate these possible task assignments against a list of requirements. Thus, one of the first steps in this research is to study what are the requirements of a suitable task assignment in an MPMS.

- RQ2: What benefits can we expect from adapting an MPMS?

Before discussing the design of an adaptation mechanism, it is crucial to understand why having this mechanism is necessary. Hence, we should present clearly how this newly added adaptation mechanism can help to improve the usability and Quality Of Service (QoS) of MPMSs. This further motivates our research efforts on developing the task-distribution-based adaptation mechanism.

- RQ3: How to design effective and efficient task assignment algorithms for MPMSs?

This research question forms the focus of this thesis. The core of a task-distribution-based adaptation mechanism is a decision-making component that can derive a new task assignment that is more suitable under the new situation. This intelligence is supported by various task assignment algorithms. By "effective", we mean the algorithm should produce a task assignment that suits the new situation the most. By "efficient", we mean, in order to compute an effective task assignment, the algorithm should not consume too much resources, e.g., CPU time and memory space. The efficiency requirement is crucial here because: (1) the algorithm may be required to execute at run-time on a resource-scarce device, and (2) the computation time of this algorithm is one of the main factors that determines the agility of task-distribution-based adaptation.

- RQ4: How to support dynamic task distribution in MPMSs?

The majority of current MPMSs are implemented as static systems in a hard-coded way. In order to support dynamic task distribution, it is required to enforce new facilities and rules on current MPMSs. Thus, this thesis designs and validates new components and protocols to enable task-distribution-based adaptation mechanisms.

- RQ5: What is the potential disruption on the telemonitoring services caused by a dynamic task distribution and how to measure this reconfiguration cost?

Dynamic system reconfiguration often has a negative impact on the on-going services provided by the system. This disruption is more prominent when the transfer of task state information is required. To guarantee that the proposed task-distribution-based adaptation mechanism does not do more harm than good, we need to understand the potential service disruption caused by reconfiguration and find ways to control the reconfiguration cost.

1.4 Thesis Structure and Credits

The rest of this thesis is structured as follows.

Chapter 2 presents the state-of-the-art in three selected fields: adaptive distributed streaming system, task assignment algorithm, and dynamic reconfiguration. Previous research on these subjects provides fundamental concepts and knowledge, and further motivates our work on task-distribution-based adaptation mechanism.

Chapter 3 studies the requirements and presents the high-level design of task-distribution-based adaptation mechanism for MPMSs. The decision is made to build the adaptation mechanism at a middleware layer. A major part of this chapter is dedicated to the formulation of task assignment problem in MPMSs. This chapter is based on [MBW+08, MvBBW+09].

Chapter 4 proposes and evaluates task assignment algorithms for situation with specific topological models of the underlying system and the processing task graph. Two methods are used in this chapter: a graph-theoretical method and a Genetic Algorithm (GA)-based method. This chapter is based on [MW07, MPW07, PMW+07].

Chapter 5 proposes and evaluates task assignment algorithms for problem in the most general form. A*-based task assignments are presented for two different problem formulations. This chapter is based on [MBP+09a].

Chapter 6 presents the design of task distribution infrastructure and validates it by means of an OSGi-based implementation. This infrastructure can support the system reconfiguration of an MPMS by means of task distribution. This chapter is based on [MWB+07, MBP+09b].

Finally, chapter 7 presents the conclusion and identifies possible future work.

Table 1.1 summarizes how the chapters are organized to address the five research questions.

Chapter	RQ1	RQ2	RQ3	RQ4	RQ5
Chapter 3	×	×		×	
Chapter 4			×		
Chapter 5		×	×		
Chapter 6				×	×

Table 1.1: *A matrix denoting how the five research questions are tackled in different chapters*

Chapter 2

Literature Review

In order to perform properly in a dynamic environment, systems should adapt itself when facing various changes in application demands and environment resources [NSN+97, FGCB98, HBL+98, LL02]. The applied adaptation methods among others include data compression [FGBA96], discarding less important information [WHZ01, TcZ07], and handover to a better network [PMW+07]. While most of the research on adaptation primarily focused on adaptation at an individual computing node [HBL+98, LL02], some work aim at a distributed adaptation involving multiple nodes to tackle the resource dynamics.

This chapter discusses the key technologies in three selected fields that are relevant with building a dynamic task distribution support for MPMS. First, adaptive distributed stream processing systems are discussed in the Section 2.1. Since an MPMS can be viewed as an adaptive distributed stream processing system as well, these earlier studied systems can present us a useful reference point when we design MPMSs. Section 2.2 reviews the research field of task assignment algorithms. Section 2.2 reviews the research work in dynamic reconfiguration.

2.1 Adaptive Distributed Stream Processing System

In [KHH05], a number of distributed adaptation mechanisms in Mobile Ad-hoc NETwork (MANET) are reviewed. The reported comparison focuses on four essential distribution aspects in MANETs: service discovery, task allocation, remote task communication, and task migration. Service discovery involves identifying and locating tasks and resources available at a particular moment. Task allocation determines how the execution of a task should be activated on a set of mobile nodes. Remote task communication covers the means for communication between distributed tasks over a wireless communication link. Task migration means the methods for transferring an executable task from one node to another. In this section, we use this same comparison framework to analyze four characteristic adaptive distributed stream processing systems, i.e., Dacia[LP01], MediaNet [HNR03], MobiPADS [CC03], and ACES [AJS+06].

2.1.1 Evaluation of Four Systems

Dacia

Reported in [LP01], DACIA (Dynamic Adjustment of Component Inter-Actions) is a framework designed to support building adaptive distributed applications in a modular fashion. Applications implemented based on the DACIA framework can adapt itself by dynamically loading new components, changing the way components interact and exchange data, moving components from one node to another, and replicating components across multiple nodes. It is argued that the DACIA framework can support transparent run-time adaptation, i.e., users should not notice the service disruption caused by component relocation and connection reestablishment. The DACIA framework consists of three main entities:

- Processing and ROuting Component (PROC). PROC is the basic building block (component) for the targeted distributed applications. It can synchronize or split input data streams, and direct them alternately to multiple destinations. PROCs are interconnected in multiple ways, according to certain rules and restrictions. Since DACIA aims at streaming applications, an Remote Procedure Call (RPC)-like invocation model is not appropriate. Thus, PROCs communicate with each other by exchanging messages through their input and output ports in an asynchronous message passing fashion. PROCs are identified system-wide using a unique identifier obtained by combining the node ID where the PROC originated and a counter maintained by the node. When a PROC is moved to a new node, its execution state is not transferred. However, a relocated PROC carries with it its data members, the messages received and not handled yet, and the state of its connections. This can be realized through, e.g., Java serialization.

- Engine. Engine is the DACIA entity at each node responsible for maintaining the list of PROCs and their connections, migrating PROCs, establishing and maintaining connections. The movement of a PROC is transparent to other PROCs. It is the Engine's responsibility to maintain the connection between PROCs after PROC relocations: messages addressed to a relocated PROC will be forwarded by the Engine to the PROC's new location.

- Monitor. Monitor is the DACIA entity that monitors the application performance, generates reconfiguration decisions, and instructs the Engines accordingly. In the reported DACIA framework, the reconfiguration decision is made by a human administrator through a command-line interface.

Based on the comparison framework, we summarize DACIA in Table 2.1.

Service discovery	"Monitor" can monitor the available tasks and application performance. However, there is no support on discovering new available nodes.
Task allocation	"Monitor" is specified as the entity that can interpret a reconfiguration plan and "Engine" has the responsibility to execute the plan. The task allocation decision is made manually.
Task communication	The data communication between PROCs are in the form of asynchronous message passing. Messages addressed to a relocated PROC will be forwarded by the Engine to the PROC's new location.
Task migration	A relocated PROC can carry its data members through data serialization. But it is not possible to transfer the execution states of a PROC.

Table 2.1: *Analysis of DACIA*

MediaNet

MediaNet [HNR03] is a distributed stream processing system designed for disaster and combat situations, where distributed mobile sensors capture and transmit real-time video and audio information to operators or analysis tools. In order to provide improved QoS in those situations where resources are often limited, MediaNet is designed to enable QoS adaptation for multiple users in two aspects. First, MediaNet allows users to specify how a streaming application should adapt under overload conditions and tries to adapt the application to maximize the user's desire. Second, in addition to using local adaptation, MediaNet can take a global view and adapt the application by dividing its operations and streaming flows among distributed network nodes. To achieve these goals, three architectural entities are defined in MediaNet.

- Continuous Media Network (CMN). Each streaming application can be represented by a set of connected data stream processing components. The model of this set of components is named as CMN and is a Directed Acyclic Graph (DAG). CMN is a conceptual model and each node in CMN represents an operation on input data streams. Some examples of these operations are data reformatting, frame prioritizing, compression, "picture-in-picture" effects, etc. In MediaNet, each user may provide a list of CMNs with different utility values to capture user's desires.
- Global Scheduler. This is a global scheduling service that can receive input

CMNs from multiple users, combine them into a single CMN, and partition and allocate it onto various network nodes. The assignment algorithm in the center of this global scheduling service works in two nested phases. In the outermost phase, the algorithm examines the utility space in a binary search. In the inner phase, the algorithm tries to find an optimal CMN assignment for a particular utility level in an exhaustive search manner.

- Local Scheduler. After the global scheduler identifies a new CMN assignment, it informs the decision to each local scheduler. The local scheduler at each node further implements the streaming operations according to the received CMN and its topological order and prescribed processing deadlines. The local scheduler also takes the responsibilities of transferring the data streams with other local schedulers on neighboring nodes, and monitoring and reporting the node's local resource usage.

Based on the comparison framework, we summarize MediaNet in Table 2.2.

Service discovery	The tasks of resource and application monitoring are performed by "Local Schedulers". In particular, each "Local Scheduler" traces the data drop rate at the application level, i.e., as an indicator of network bandwidth changes, and periodically informs the "Global Scheduler". MediaNet does not support discovering new nodes.
Task allocation	Although the binary search applied during the outermost phase can improve the task assignment algorithm performance, the limitation is in the inner phase where an exhaustive search is performed.
Task communication	Continuous multimedia streams, e.g., audio and video, are transferred between CMN components.
Task migration	A control protocol is defined to allow old and new configurations to run in parallel until the old configuration can be removed. Thus, the reconfiguration is not performed by task migration and execution states are also not transferred.

Table 2.2: *Analysis of MediaNet*

MobiPADS

Reported in [CC03], MobiPADS (Mobile Platform for Actively Deployable Service) is a middleware that supports context-aware mobile applications by enabling active service deployment and reconfiguration. This design goal is very similar to the one of our MPMSs. However, instead of a heterogeneous network, MobiPADS is designed to work in a client/server environment, where the server node resides in a wired network and the client node is a mobile device. By dynamically offloading part of computations to the server node [GNM+03, OYL06], the system can optimize the resource usage of a mobile application at the client node. Service chains are defined as entities providing certain functionalities for applications running on top of MobiPADS. The base unit of a service chain is named as "mobilet", which is explained below with a set of system components in MobiPADS:

- Mobilet: This is a re-allocable service object and a base unit that can form service chains. Mobilets exist in pairs: a master mobilet resides at the MobiPADS client and a slave mobilet resides at the MobiPADS server. The master mobilet instructs the slave mobilet to share a major portion of the processing burden, i.e., offloading the computational tasks.

- Configuration Manager: This component is responsible for negotiating the connection between the client node and the server node. It also has a service controller for initializing, interconnecting, and managing the mobilets.

- Service Migration Manager: This component manages the process of importing and exporting mobilets between the server node and the client node. It also cooperates with the service directory to activate, store, and keep track of the changes made to the active mobilets.

- Service Directory: The service directory records all the known service types. The mobilet's are stored in a service repository, which is used for service activation and service migration.

- Event Register: The event register allows objects to register for events. Event sources include various changes in network status, machine resources status, and connectivity status.

- Channel Service: The channel service provides virtual channels for mobilet pairs to communicate. Instead of opening separate TCP connections for each message, messages are multiplexed into a single persistent TCP connection, which then eliminates the overheads of opening new TCP connections and avoids the slow-start effect on overall throughput.

Two parts of MobiPADS framework were discussed intensively in [CC03]: contextual event model and dynamic service reconfiguration. The event model is proposed to provide effective means to monitor the changes in the environment and to inform the relevant information to all of the interested entities. This contextual event model is referred in our work to determine the context information relevant for MPMSs. Dynamic service reconfiguration in service chains is realized through three steps: service deletion, service suspension, and service addition. The reconfiguration time is both modeled mathematically and measured with experiments. It is observed that, in MobiPADS, a relative simple reconfiguration, i.e., two times deletions and two times additions with mobilets in the size of 5KB over a 1Mbits/s connection, takes about 1.1 seconds when mobilet's source code is loaded locally and about 2 seconds when the source code is loaded remotely.

Based on the comparison framework, we summarize MobiPADS in Table 2.3.

Service discovery	Service discovery is supported through the "Service Directory" and "Event Register". The proposed model of context information further provides detailed knowledge about the environment. Node discovery is not supported.
Task allocation	No task assignment algorithm is proposed. Instead, "Configuration Manager" plans a reconfiguration based on a set of pre-defined policies on which service chain should be used under which situation.
Task communication	All data communication between pairs of master mobilets and slave mobilets are aggregated and transferred through a TCP connection.
Task migration	Task migration is achieved by offloading a major portion of the processing burden from a master mobilet to a slave mobilet. It is possible to share the execution states between a mobilet pair.

Table 2.3: *Analysis of MobiPADS*

ACES

Presented in [AJS+06], ACES (Adaptive Control for Extreme-scale Stream processing systems) targeted at the problem of extreme-scale data mining, e.g., continuous queries over sensor data and high performance transaction processing. ACES is a two-tiered approach for adaptive and distributed resource control. The first tier

determines the resource assigned to Processing Elements (PEs) at each individual node. Second tier decisions are made dynamically by the distributed nodes. At each node, the input and output rates and the instantaneous processing rate of PEs are adjusted dynamically with the goal of stabilizing the system in the presence of burstiness. To support this two-tiered approach, the authors defined three architectural entities.

- PE: PE is the basic building block of streaming applications. In ACES, every PE is labeled with three variables: the average input data amount, the average output data amount, and the normalized average CPU allocation. ACES defines a so-called "max-flow" policy to synchronize the data processing rate when a PE's output stream is read by multiple downstream PEs. The "max-flow" policy is to set the output data rate of the parent PE equal to the maximal input data rate of all downstream PEs. The rationale is to allow PEs enough data to fully utilize their allocations despite the fact that some PEs may have to drop certain input data frames.

- Meta Scheduler: A meta scheduler is the entity to perform the global optimization such that the weighted throughput is maximized. This is achieved by maximizing a weighted summation of all PEs's output rate utility functions, which are strictly increasing and concave. Any concave optimization algorithm can be used for this purpose. Two constraints are taken into account. First, the combined CPU allocations of all the PEs on a node should be less than the available CPU cycles. Second, the output rate of a PE is no less than any input rate of its downstream PEs according to the "max-flow" policy.

- Resource Controller: The presence of burstiness, caused by relatively large chunks of data, is very common in streaming systems. Adding buffers is an unavoidable but sometimes expensive solution. In order to use the buffer space wisely, a local resource controller at each node can perform some local adjustment on the input and output rates and the instantaneous processing rate of PEs in a distributed manner. The goal is to maintain stability of the system, and avoid loss of processed data due to buffer overflow.

Based on the comparison framework, we summarize ACES in Table 2.4.

2.1.2 Concluding Remarks

As illustrated earlier, the current distributed adaptive systems handle the issues like performance monitoring, task communication effectively. However, in order to build an adaptive MPMS, there are still problems need to be addressed. First, the

Service discovery	"Resource Controller" is responsible for monitoring the on-going application performance. When a burstiness is detected, it can adjust the data transmission rate in a distributed manner. Node discovery is not supported.
Task allocation	"Meta Scheduler" determines the PE assignment when the applications are deployed or periodically to support changing workload and resource availability.
Task communication	The data communication between PEs are streams. The input and output rates and the instantaneous processing rate of PEs can be adjusted dynamically.
Task migration	After PEs are initialized at each node, migration is not supported during the application execution.

Table 2.4: *Analysis of ACES*

discovery of new resources, e.g., a newly joined device, is often not supported by these adaptive systems. Thus, it is unlikely that the current adaptive systems can fully benefit from emerged new resources. Second, although some systems, e.g., MediaNet and ACES, have a dedicated decision-making component to compute the optimal system configuration, their embedded decision-making algorithms certainly have room for improvement. For example in MediaNet, an exhaustive search is performed in the inner phase. Third, the problem of execution state transfer and reconfiguration cost during task migration are often omitted or not sufficiently addressed. Hence, while learning the features from the current systems, we should make sure that the lacking features to be addressed in MPMSs properly.

Centralized vs. Decentralized Approaches

As observed in the reviewed systems, an adaptation in distributed systems is carried out by local adaptation agents at various system nodes. Although these adaptation actions are done in a decentralized fashion, the adaptation plan can be either constructed at a centralized entity or composed by separate decisions that are made by distributed entities.

The advantage of centralized approaches is clear: a centralized entity with complete knowledge on the system is capable of constructing a consistent and optimal adaptation plan. However, their drawbacks can be recognized easily too. The most notable ones are the risk of single-point of failure and the problem of scalability.

The decentralized approach has been used in building distributed adaptive systems, e.g., [Smi80, BB04]. Due to the nature of decentralized approaches, these systems often do not produce global optimal plans and potentially incur significant overhead during the negotiation process that is critical on preventing the use of inconsistent plans [Lo88, GA90].

Our targeted MPMSs are not large-scale systems, thus the scalability is not a crucial concern. Second, the risk of single point of failure can be prevented in the system deployment, e.g., by introducing duplicated decision-making entities. Third, the critical missions carried by MPMSs usually have a high expectation on the system performance. Based on these, the approach selected in this thesis is a centralized one. All four adaptive systems surveyed in this section also adopted a centralized decision-making approach.

2.2 Task Assignment

Task assignment is an optimization problem that exists in the field of parallel computing [NT93], grid computing [CDK+04, MKK+05], System-on-Chip design [HM05], distributed database [PLS+06], and in-network processing for wireless sensor networks [GTE07]. Except for some special cases, finding the optimal task assignment is a NP-hard problem [NT93]. In this section, we first try to get a better understanding on this problem and then review a number of task assignment algorithms in two categories: *exact* and *heuristic*.

2.2.1 Task Assignment vs. Task Scheduling

In various work, task assignment problem has been referred to as allocation [SWG92], partitioning [Bok88], matching [BSBB98, Ds02] or mapping[1][Bok81]. It also appeared in literature on task scheduling and sometimes is regarded as a scheduling problem [CK88, KA99]. In this thesis, we present a clear separation[2] between the problem of task assignment and the problem of task scheduling as follows.

In a distributed computing system with multiple processors[3], when given an application consisting of multiple distributable tasks, we need to take the following two steps in order to deploy and execute the application:

[1] This term is sometimes used to address the combined problem of task assignment and scheduling [EW97, BSBB98]

[2] A similar distinction can be found in [GTE07]

[3] We use the term of "processor" in this section instead of "node". The reason is twofold. First, this is in line with most literatures on task assignment which are originated from the field of parallel computing. Second, the term of "node" could be easily confused with a node in a graph in this section.

1. Assigning tasks to processors subject to certain resource limitations and constraints imposed by the system.
2. Determining the start time of tasks on each processor.

In the first step, we need to solve the problem of task assignment and in the second step, it is a problem of task scheduling. In MPMSs, the member tasks process streaming data and all start the execution once the application is activated. Thus, scheduling is not a relevant problem in our system.

2.2.2 Models of Task Assignment

A task assignment problem can be modeled from three aspects as summarized in [NT93]:

1. A set of networked processors and their resource characteristics - resource model.
2. A set of tasks, their communication patterns and their demand characteristic - task model.
3. The cost function to determine the quality of an assignment - cost model.

Resource Model

The resource model is often represented as a graph where processors are modeled as vertices and the communication channels between processors are modeled as edges. The detailed characteristics of this graph vary in the following dimensions:

- *Heterogeneity*: Some studies [YC94, IB95, LS97, Woe01] assume the resources are homogenous, i.e., all processors and channels are identical. Other studies are based on a more general model where processors and channels are heterogeneous, e.g., [Sto77, Bok88, NO91, SC90, HL92, KA98].
- *Topology*: Based on the observations from various systems, different topologies of processor network were studied, e.g., chain [Bok88, NO91, SC90, HL92, YC94, IB95, Woe01, PA04], star [Bok88, IB95], tree [LS97], array [LS97], fully-connected network [Lo88, SCS+04, UAKI06, XQ08, BR08], and arbitrary network [KA98].
- *Direction*: So far, most work study processor networks with only symmetric channels. In these networks, each channel can transfer data streams on both directions with the same characteristic. Some work deal with asymmetric channels and model the channel directions explicitly, e.g., in [HM05, AAH05].

- *Relaying support*: Most studies assume a non-fully-connected network and do not consider the possibility of data relaying by processors. This means that they require two connected tasks to be assigned to either the same processor or two adjacent processors. This property is defined as the contiguity constraint in [Bok88]. Some work permit the existence of relaying processors in their resource models, e.g., [Fra96, LS97].

Task Model

There are two commonly used models for representing a distributed application: the Task Interaction Graph (TIG) and the Task Precedence Graph (TPG) [KA99]. TIG is an undirected graph where nodes represent tasks and edges represent bi-directional communications between two tasks. TPG is often a DAG where nodes also represent tasks and directed edges represent task precedence relationships. Based on the observations from various systems, some specific applications can be modeled as graphs in a special topology, e.g., a chain or a tree.

Cost Model

The following performance measures have been used to define the cost function of task assignment:

- *Abstract total cost*: This is the most common measure used in task assignment research [Sto77, LS97, KA98]. It is a combined cost of computation (caused by task execution at processors) and communication (caused by data transmission at network channels). The goal of these task assignment algorithms is to minimize the total cost.
- *Data throughput*: The overall data throughput has also been used as the objective function [ST85, Bok88, NO91, SC90, HL92, YC94, IB95, Woe01]. These studies try to minimize the task load on the most-heavily-loaded processor and it is sometimes refereed as *minimax* criterion in the literatures.
- *End-to-end delay*: This is the elapsed time between a specific source task receives a frame data input and a specific sink task produces the corresponding processed result of this particular data frame. In [CNNS94], this measure is also termed as the single data set's response time. This measure resembles the most important objective function in task scheduling problem, the *makespan*, which is the total execution time of a schedule.
- *Energy consumption*: Energy consumption is a critical issue especially when dealing with a system containing mobile nodes. Various algorithms [SCS+04,

AAH05, GTE07, LV07, XQ08] have been proposed to find the optimal task assignment that minimizes the total energy consumption. In [YP05], the authors argue that the focus on minimizing total energy consumption may lead to heavy use of the most energy-effective device regardless of its remaining energy and propose to use maximizing the system battery lifetime as the objective function.

- *Reliability*: Reliability is the probability that a failure will not occur on any processor or any channel when they are used for executing an application. The failure chance of every processor or channel depends on its accumulative execution time, and this time is further determined by the number and the type of tasks assigned to it. Thus, by adjusting the task assignment, the system reliability can be controlled. In [SWG92, Fra96], authors proposed an A*-based algorithm to determine the optimal task assignment to maximize the reliability.

2.2.3 Exact Approaches

In this section, several exact task assignment algorithms are reviewed. These algorithms can find the optimal task assignment in polynomial-time due to some restrictions on the model of tasks or processors that make the problem tractable.

Two-Processor Task Assignment

Stone [Sto77] proposed a polynomial-time solution based on the Max-flow/min-cut theorem to solve the two-processor task assignment problem: To assign an application consisting of m interacting tasks onto two connected processors so as to minimize the total communication and computation costs. While an exhaustive search suffers a time complexity of $O(2^m)$, Stone's solution can identify the optimal assignment with a bounded time complexity of $O(m^3)$.

The model of this problem is illustrated in Figure 2.1. An application is modeled as TIG. The edges are weighted with the communication cost. The computation cost of executing the tasks on each processor is given in the table. The symbol of "∞" indicates an infinite cost, which means that this task cannot be executed on the given processor. It is assumed that the intra-processor communication cost is negligible. Every assignment of tasks onto these two processors results in a different total cost. For example, when task "B" is assigned to processor "N1" and all the other tasks are assigned to processor "N2", its total assignment cost is the sum of (1) the execution cost at "N1", i.e., 2, (2) the execution cost at "N2", i.e., $10 + 4 + 3 + 2 + 4$, and (3) the communication cost between "N1" and "N2", i.e., $6 + 8 + 12 + 3$.

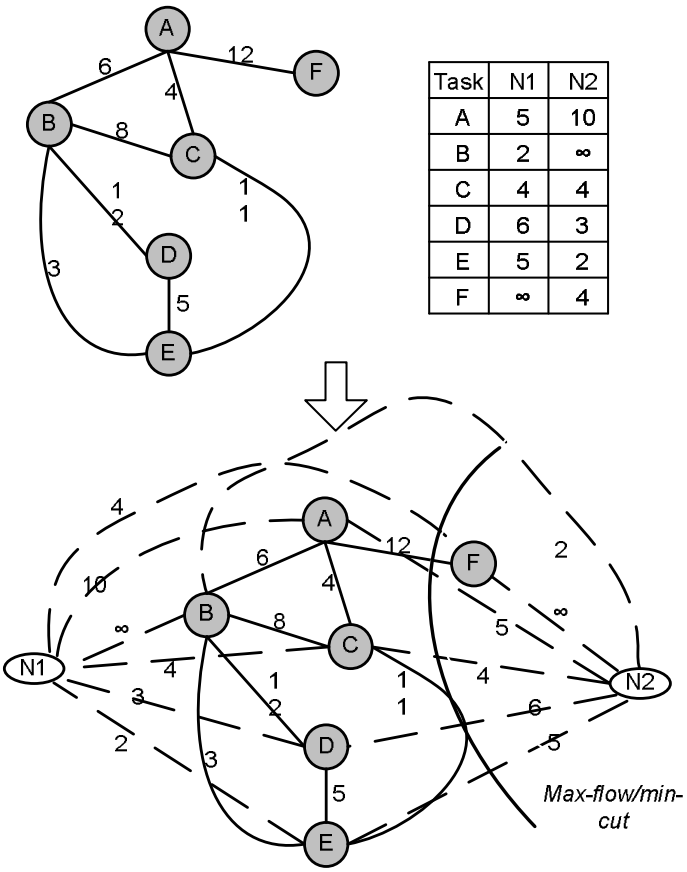

Figure 2.1: *Stone's method for two-processor task assignment*

The proposed solution by Stone is summarized as follows:

1. In the task graph, add two nodes "N1" (source) and "N2" (sink) that represent two processors. Then connect these two newly added nodes with every task node. The weight of the edge connecting to "N1" is the cost of executing the corresponding task on "N2", and the weight of the edge to "N2" is the cost of executing the corresponding task on "N1". (Such a reversed weighting is intentional.)

2. The modified task graph exemplifies a commodity flow network. Each cut between the source and the sink of this graph partitions the nodes into two subsets, with "N1" and "N2" in different sets. The weight of each cut equals

the sum of the weights of all cutting-through edges. Due to the intentionally assigned weights, this sum equals to the total cost of the corresponding task assignment.

3. Now the problem of finding the optimal task assignment is transformed into finding the cut with minimum weight (min-cut). Several graph theoretic algorithms are available to solve this problem in polynomial time. A standard "push-relabel algorithm with FIFO node selection rule" [CLRS01] can find the "min-cut" in time $O(|V|^3)$ and a latest design [SW97] can provide a simpler solution in time $O(|V||E| + |V|^2 \log |V|)$. For example, as shown in Figure2.1, the optimal assignment is to assign task "F" to "N2" and the rest to "N1".

In the same paper, Stone tried to extend this two-processor algorithm for solving the problem on n-processor ($n \geq 3$). It appears natural that the task assignment problem on n-processor can be reduced to a number of two-processor problems, i.e., adding n distinguished (processor) nodes representing n processors into the task graph and finding the min cutset corresponding to the optimal assignment. However, the result was negative. The main difficulty identified by Stone is that "it is easy to show that a task node associated with a particular distinguished (processor) node in a minimum n-processor cutset fails to be associated with that (processor) node by a two-processor cut". Although the extension towards solving n-processor problem failed, Stone's two-processor algorithm laid important foundation for the later heuristic approaches [Lo88, KLZ97].

Bokhari extended Stone's method on finding the optimal task assignment in two-processor network with the consideration of task relocation cost [Bok79]. In this model, each task may have different characteristic in different execution stages. Therefore, it is plausible to relocate the task at the end of each running stage if the gain from relocating tasks can outweigh the relocation cost.

Chain-Chain Task Assignment

In typical streaming systems, a fixed sequence of tasks is executed onto a continuous series of data frames. For a better performance on the data throughput, it is common to set these tasks as pipelined operations on a chain of processors. Thus, this group of problems can be modeled as the assignment of a task-chain onto a processor-chain as shown in Figure 2.2. The tasks and processors are heterogeneous and the computation time of executing one data frame by every task on each processor are known. The channels between processors are heterogeneous and the communication time of transferring one data frame exchanged between two connected tasks on these channels are known. In order to determine the optimal task assignment with

a minimal bottleneck processing time (thus the highest data throughput), Bokhari proposed a polynomial time algorithm [Bok88] with the following two assignment constraints:

- Any two adjacent tasks have to be placed on the same processor or two adjacent processors (the *contiguity constraint*).
- Every processor has to be assigned with at least one task (the *full-use constraint*).

This chain-chain task assignment algorithm works as follows:

1. Building the assignment graph: It is a layered graph that contains all information about possible sub-assignment of tasks, and its associated computation and communication time. As illustrated in Figure 2.2, each layer corresponds to a processor and the label on each node corresponds with a possible task-chain to be assigned. Given an m-task chain and an n-processor chain ($m > n$), the number of nodes per layer is of the order of $O(m^2)$ and the number of nodes in the graph is of the order of $O(m^2n)$. Each node has at most m edges connected to it, thus the number of edges is of the order of $O(m^3n)$. Any path connecting node "S" to node "T" corresponds to an assignment of tasks to processors. For example, the thick path correspond to the example assignment . It is possible to reduce the size of an assignment graph by deleting the node which reflects an invalid assignment, e.g., against local memory limit, and the edges incident on it.

2. Labelling the edge with the associated computation and communication time: In layer k, each edge pointing downwards from node "$< i, j >$" is first weighted with the time required for processor k to process tasks i through j on one data frame. (Intra-processor communication time is negligible.) Then, to the weight on the edge from "$< i, j >$" in layer k to "$< j + 1, l >$" in layer $k + 1$, a communication time between task j and $j + 1$ over the link between processor k and $k+1$ is added. The value of this communication time thus represents both the data frame size transmitted between task j and $j + 1$ and the speed of link between processor k and $k + 1$.

3. Finding the minimal bottlenecked path: A variant of Dijkstra's shortest path algorithm can be used to find the optimal bottleneck path in $O(m^4n^2)$ time. Based on the special layered feature of the assignment graph, a faster procedure based on dynamic programming is also proposed. This improved procedure visits each edge in the assignment graph exactly once, thus it has a time complexity of $O(m^3n)$.

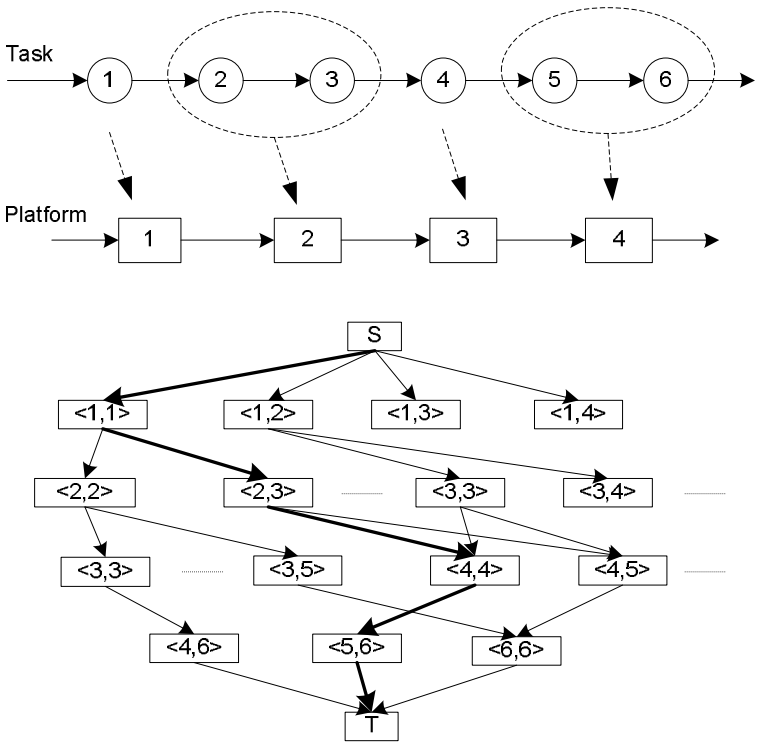

Figure 2.2: *Chain-chain*

Three improved algorithms were further reported on this chain-chain assignment problem with the same model on task, resource and cost and the same contiguity constraint. In [NO91], the authors modified the assignment graph proposed by Bokhari by adding $(n-2)$ extra layers to reduce the number of edges (and thus the algorithm's time complexity) from $O(m^3n)$ to $O(m^2n)$. Independently, a dynamic programming based procedure is proposed in [HL92] to find the optimal assignment with the same time complexity of $O(m^2n)$. This procedure works by recursively computing the bottleneck processing time f_{ik} when the first i tasks are optimally assigned to the first k processors. In [SC90], the authors proposed a new method to construct the task assignment graph with the number of edges of the order of $O(m^2n)$ but relax the full-use constraint.

When the model is further constrained to homogeneous processors and channels, even faster algorithms can be found. In [IB95], a "probe" function is proposed to check the possibility of assigning a task-chain to a processor chain while keep-

ing the bottleneck processing time below a given value. Then using a binary search on a set of testing values, the authors can identify the optimal task assignment. The overall time complexity of this "probe" based approach is $O(mn \log m)$. Based on dynamic programming, an $O(mn \log m)$ algorithm is reported in [YC94] and an $O(mn)$ algorithm is reported in [Woe01]. Based on a model that further abstracts the problem as partitioning a sequence of m real numbers into n intervals optimally, a $O(m(m-n))$ algorithm is reported in [OM95].

Tree-Star Task Assignment

In [Bok88], the problem of assigning a tree-structured task graph onto a single-host, multiple-satellite processor network is studied. This is a model motivated by many industrial process monitoring systems where information from multiple sensors are continuously collected by small satellite processors and transmitted to a large central host for processing. The resource model is represented as a star (Figure 2.3) where all the satellite processors and the channels connecting satellite processors to the host processor are homogeneous. The task model is represented as a tree where the data flows from leaf tasks to the root task. The goal of this task assignment problem is similar to the chain-chain problem: to find the task assignment that has the highest data throughput, thus to minimize the largest execution time among all satellites and host. An example of this tree-star task assignment is shown in Figure 2.3.

Furthermore, the following four assignment constraints are enforced:

- The root task is always assigned to the host processor.
- Once a task is assigned to a satellite, all its children tasks are also assigned to the same satellite.
- If two tasks are assigned to a satellite, their lowest common ancestor is also assigned to the same satellite.
- There are as many satellites as there are leaf tasks and we may choose not to use them if the optimal assignment indicates so.

To address this specific problem, Bokhari introduced a method of constructing an assignment graph based on the task tree. This generated assignment graph can represent all the eligible tree-star task assignments. Every edge of the assignment graph is further labeled with two weights that contain the information about the potential work load on both the host and satellite processors. A path search algorithm (SB algorithm) with a time complexity of $O(m^2 \log m)$ is proposed for searching the

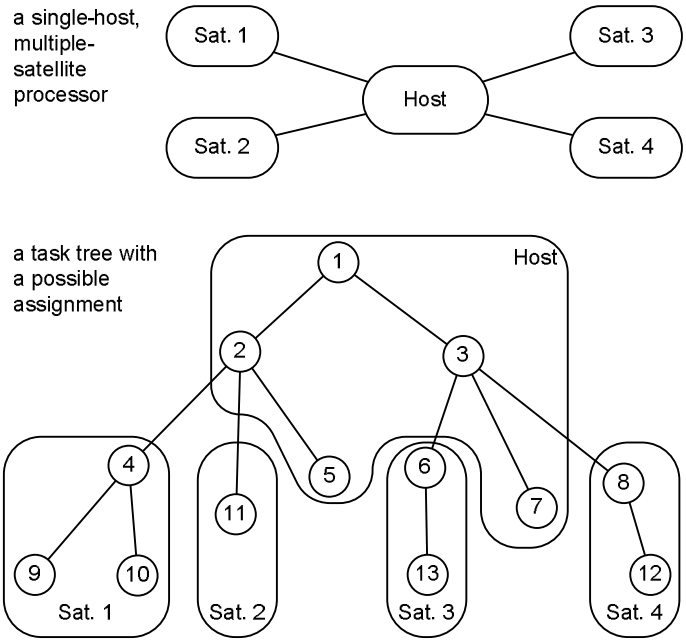

Figure 2.3: *A tree-structured program partitioned over a host-satellite system*

optimal path based on these two weights. This optimal path indicates the optimal task assignment of tree-star problem.

A*-based Task Assignment Algorithms

In case of smaller problem size, it is computationally feasible to find the optimal task assignment by A*-based exact algorithms. Shen and Tsai [ST85] are the first researchers to apply A*-algorithm in the task assignment problem in distributed systems. In their paper, a task assignment is defined as a weak homomorphism between a task graph and process graph, i.e., two adjacent tasks are required to be assigned to either the same processor or two adjacent processors. The objective of their algorithm is to minimize the largest total computation and communication costs at a processor. In [RCD91], authors proposed an ordering method to create a better task search order that can reduce the number of A* search tree nodes and thus increase the algorithm speed and reduce the memory requirement. In [KA98], authors proposed two techniques to further enhance the A*-algorithm performance. The first technique is to generate a random task assignment and use the corresponding cost as a pruning criterion to reduce the size of search tree. The second technique

is to divide the search tree into parts and speed up the algorithm by parallel processing. While all these existing work on A*-algorithm use an abstract communication and computation cost as the performance measure, we are interested in multiple performance measures that are more relevant to the user and network context in MPMS.

2.2.4 Heuristic Approaches

A more general form of task assignment problems deals with a n-processor network ($n > 2$). The topological model of this processor network is a fully-connected graph with symmetric channels. This group of task assignment problems has been identified as NP-hard [NT93]. In this section, we review several heuristic approaches on this general form.

Lo's "Grab-Lump-Greedy" Algorithm

Lo [Lo88] introduced a heuristic algorithm of assigning a task graph (containing m tasks and e edges) to a n-processor ($n \geq 3$) network. The assignment cost is modeled as the total execution and communication cost. The task model is TIG and it is assumed that the tasks' execution costs and communication costs are known. The resource model is heterogeneous processors, homogenous and symmetric channels, fully-connected processors resulted by a systemwide message-passing mechanism, and no relaying processor. This heuristic algorithm consists of three phases: (1) Grab, (2) Lump, and (3) Greedy. The complexity of these three phases are $O(m^2ne\log m)$, $O(m^2e\log m)$, and $O(e)/O(m^2n)$ respectively. The "Grab" phase follows a "divide and conquer" approach and works as follows:

1. In each round, we select one particular processor and combine all the other processors as a "super processor". Now, this is transformed to a 2-processor task assignment problem (Figure 2.1) and a "min-cut" algorithm can be applied to find an optimal assignment. This routine is repeated to every processor and this is to let each processor grab its favorite tasks. In [Lo88], it is proved that no two processors will grab the same task.

2. When all the processors are visited, a round of "Grab" is finished. Then, the task graph is reconfigured by removing all the tasks assigned in this round and updating the remaining tasks' execution cost.

3. The "Grab" phase continues until no further assignment of tasks occurs in a round.

After the "Grab" phase, if there is no task left, the current assignment is the optimal n-processor assignment. Otherwise, the "Lump" phase is called to check whether it is worthwhile to assign all remaining tasks to a single processor. If the "Lump" phase still can not assign all the tasks, the "Greedy" phase is invoked. In the "Greedy" phase, the remaining tasks which have high communication cost in between are merged into clusters and these task clusters are assigned to the cheapest processor.

Lo further introduced the concept of "interference cost", which reflects the penalty of assigning too many tasks to the same processor and archives an even load among processors.

"Max Edge" Algorithm

In [KLZ97], a heuristic algorithm, i.e., "Max Edge", with a time complexity of $O(m(m+n)^2)$ is proposed. This algorithm works in the same problem model but is reported to outperform Lo's "Grab-Lump-Greedy" algorithm. This algorithm works as follows:

1. Transform the given TIG (G) and n available processors into a graph termed as G': A special node for each processor is added into G and is connected with every task nodes with an edge and the edge weight $c_{ik}^{'}$ is defined a $c_{ik}^{'} = \frac{\sum_{p=1}^{n} e_{ip} - e_{ik}}{n-1}$, where e_{ip} is the computation cost of task i at processor p. $c_{ik}^{'}$ can be viewed as the average computation cost of assigning task i to processors except for k. In G', the weights of the edges ($c_{ij}^{'}$) connecting two task i and j nodes remain the same as G.

2. Iteratively remove the edge with largest weight $c_{ij}^{'}$ in $G'(V', E')$.

   - If i and j are tasks, then group them as a new task k and set $c_{kl}^{'} = \max\{c_{il}^{'}, c_{jl}^{'}\}, \forall l \in V'$.
   - If i is task and j is processor, allocate i to j, delete edges $(i, p), \forall p \in P$. And set $c_{lj}^{'} = \max\{c_{li}^{'}, c_{lj}^{'}\}, \forall l \in T$.

   This step is repeated until all tasks are allocated.

The complexity of this "Max Edge" algorithm is $O(m(m+n)^2)$: the repeat loop is executed at most $O(m+n)$ times and the edge weight updates in each loop require $O(m(m+n))$.

"Multilevel" Algorithm

In [UAKI06], a set of "Multilevel Clustering" heuristic algorithms are proposed. This algorithm works in the same problem model as Lo's "Grab-Lump-Greedy" algorithm and Kopidakis's "Max Edge" algorithm. The multilevel approach has been used successfully in graph and hypergraph partitioning problems [HL95, KK95, KK96]. There are three phases in the multilevel approach: clustering, initial solution, and refinement. In graph and hypergraph partitioning problems, the processors are assumed to be homogeneous. Therefore, the multilevel techniques developed for the graph partitioning problems can not be applied directly to task assignment problems. To tackle the task assignment problem, the proposed "Multilevel Clustering" heuristic algorithms work as follows:

1. Clustering phase. In this phase, the given TIG $G = G_0 = (T_0, E_0)$ is transformed into a sequence of smaller TIGs $G_1 = (T_1, E_1), ..., G_k = (T_k, E_k)$, where $|T_0| > |T_1| > ... > |T_k|$. This is to cluster disjoint subsets of tasks of G_l at level l into supertasks such that each supertask in G_l forms a single task of G_{l+1} at level $l + 1$. The execution times of each task of $G_l + 1$ is the sum of execution times of its member tasks in G_l. The edge set of each supertask is set to the weighted union of the edge sets of its member tasks, where the internal edges are deleted. Four different clustering algorithms are proposed in [UAKI06]. Among them the "Matching-based clustering" offers the best performance and it works as follows: For each edge (i, j) in G_l, a clustering profit for tasks i and j is calculated. Then, the edges with nonnegative clustering profits are visited in the descending order of clustering profits. If both of the incident tasks are not clustered with other tasks yet, then these two tasks are merged into a cluster. At the end, unmatched tasks remain as singleton clusters for the next level.

2. Initial solution. At the coarsest level, a simple initial assignment algorithm is used to assign every task or supertask to it's favorite processor according to some assignment criteria.

3. Refinement (Uncoarsening phase). At each level l, assignment A_l found on the task set T_l is refined to an assignment A_{l-1} on the task set T_{l-1}: The member tasks of each supertask in G_{l-1} are assigned to the processor to which their clustered supertask is assigned in G_l. In the decomposed T_{l-1}, a maximum reassignment gain, the gain of assigning task i to other processors instead of the already determined one, for each task $i \in T_{l-1}$ are calculated. By calculating, updating and sorting these reassignment gains for each task for several rounds. The task assignment A_{l-1} on the task set T_{l-1} at level l can be refined.

Alsalih Algorithm

In [AAH05], the resource model is represented as two matrixes: (1) $CommT$ contains estimated communication delay among processors, where $CommT(i, j)$ is the amount of time required for one unit of data to travel from p_i to p_j. (2) $CommE$ gives estimated communication energy consumption among processors, where $CommE(i, j)$ is the amount of energy consumed by one unit of data to travel from p_i to p_j. This includes the energy consumed by source device, destination device and overhearing devices.

Although it is not mentioned explicitly in [AAH05], this proposed matrix representation of processor network can model the directions in processor networks and support the existence of relaying devices. However, it has the following limitations:

- The proposed matrix representation can model the reachability between devices, i.e., it is actually a "transitive closure" matrix of the processor network. However, there are potentially multiple directed paths connecting two devices. The current matrix representation can not model and distinguish these paths.

- In [AAH05], "total consumed energy" is used as the objective function. The represented energy consumption information, i.e., $CommE$, only contains the total energy consumed by one unit of data to travel from p_i to p_j. But it can not model the energy consumption on each device that is involved in the data transmission between p_i to p_j. The lacking of this energy consumption information on each device makes it is impossible to estimate the system battery lifetime.

- A heuristic list scheduling algorithm is applied for task assignment: in every step during the task assignment phase, a new task is examined and assigned only to a processor that can satisfy the reachability constraint. However, due to its greedy approach, this algorithm can not guarantee to produce an optimal assignment.

2.2.5 Concluding Remarks

Earlier work [NT93] has shown that finding the optimal task assignment is an NP-hard problem except for several special cases. Our problem in MPMS belongs to the same category and we expect that only restricted forms of the general problem can be solved in polynomial time. Furthermore, the task assignment problems dealt within MPMSs to be more complicated given the following facts:

- The underlying heterogeneous m-health platform has to be considered as a general graph with asymmetric links while most existing models consider a fully connected network with homogenous symmetric connections [MCC04, AJS+06, UAKI06]. Hence, one additional requirement of task assignment algorithms in MPMSs is that the direction of data flow in task graph should be aligned with the direction in the m-health platform.

- The possibility of relaying data stream by devices in an MPMS further complicates the representation of a task assignment. In existing work, relaying data stream has not been a main concern since fully connected networks are often considered. In [Fra96, LS97], authors considered the possibility of relaying data streams by devices. However, the performance estimation of a task assignment does not include the extra resource consumption caused by relayed data streams at a relaying device.

- The majority of the existing algorithms are proposed based on a rather abstract cost model, e.g., the total computation cost and communication cost [Lo88, NT93, UAKI06]. However, when dealing with a real application in an MPMS, we are interested in a measure that reflects more realistically the system performance. In other words, we should look at measures at a much detailed level, e.g., measures related with energy consumption, delays, etc.

2.3 Dynamic Reconfiguration

Dynamic reconfiguration is a technique to improve service availability since it can update a running system without taking it off-line [Weg03]. This technique has been applied to deal with changing user requirements [GN06] and underlying resources [AJS+06] in distributed streaming systems. In [ZL07], the existing dynamic reconfiguration mechanisms are divided into two main categories, stateful (general purpose) mechanisms and stateless (particular scenario) mechanisms.

Stateful mechanisms can preserve the system *correctness* without the knowledge of detailed application information. As defined earlier [Gou99, Weg03], a system after a reconfiguration is said to be correct if:

1. The system satisfies its structural integrity requirements,

2. The entities in the system are in mutually consistent states, and

3. the application state invariants hold.

The latter two conditions require the dynamic reconfiguration mechanism to wait or drive the system into a consistent/safe state and then transfer states between components. Blocking the relevant components or communication paths are commonly used for this purpose. Unavoidably, these operations will have their impact on the system and its provided services.

On the other hand, Stateless mechanisms are designed for some particular application scenarios and can be viewed as optimized versions of stateful mechanisms [HNR03, ZL07]. The reconfiguration algorithm [Mit00] designed for the pipe-and-filter architectural style is a good representative example. The main assumption in those systems is that components are stateless. Therefore, the state transfer is not required and the impact on service performance can be minimized.

In this section, we review four stateful mechanisms and two stateless mechanisms. One of the stateful mechanisms is regarded as the fundamental work in this field [KM85] and thus is reviewed separately first.

2.3.1 Fundamentals

Prior to the seminal work of Kramer and Magee [KM85, KM90] on dynamic reconfiguration, the underlying support mechanisms for change (software component creation, binding, and deletion) in distributed systems are readily available, but little studies are carried out on how such dynamic changes should be specified, managed, and controlled. To tackle this challenge, Kramer and Magee present a model of dynamic change management (Figure 2.4) which separates the concerns over specifying system structural changes from the concerns over building application components that can support these changes. This separation of concerns permits the formulation of general rules for defining change specifications without the need to consider application state, and the specification of application component behaviors without prior knowledge of the actual structural changes.

Kramer and Magee defined a set of objectives (requirements) of a dynamic reconfiguration management system. The principles embodied by these objectives have influenced a number of subsequent work, e.g., [Mit00, Weg03].

- *Changes should be specified in terms of system structure.* In software system that is constructed in a modular way, e.g., a component-based system, it is both possible and pragmatic to support the change at a module/component level. Attempting to perform changes at an even lower programming level is impractical due to the level of detail involved, whereas the change at the more loosely-coupled component level can be easily understood and controlled.

- *Change specifications should be declarative.* It is the configuration management

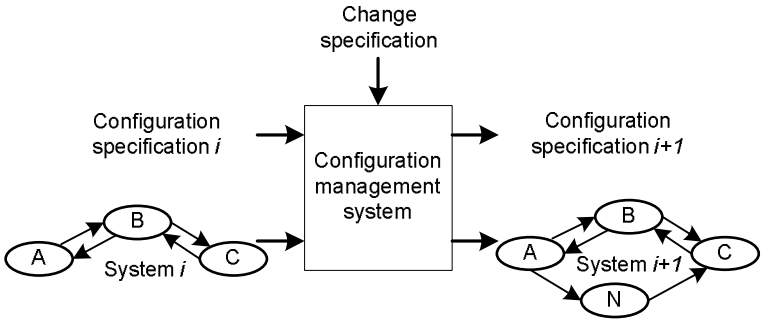

Figure 2.4: *The fundamental model of dynamic change management*

system, not the user, who should determine the specific ordering of actual change operations applied to the system, i.e., how the change will be carried out.

- *Change specifications should be independent of the algorithms, protocols and states of the application.* In order to provide generic configuration management, there must be no dependencies between the application and the configuration management system.

- *Changes should leave the system in a consistent state.* A consistent state is informally defined as one from which the application may continue normal processing, rather than progressing towards an error state. The application may of course pass through inconsistent states during the progress of a configuration change.

- *Changes should minimize the disruption to the application system.* Many systems cannot be shut down or disrupted for extended periods, so changes should be executed promptly and only interfere with those parts of the system actually affected by the change.

2.3.2 Stateful Mechanisms

Goudarzi

In [Gou99], Goudarzi identified several problems existed in Kramer and Magee's approach:

- It places a heavy burden on the application programmer who must write all nodes of the system such that they respond correctly to the command to drive

to a passive state. Kramer and Magees approach thus requires substantial effort from the application developer to make the system reconfigurable, and it also requires expertise from the application developer.

- Since all entities capable of initiating a transaction directly or indirectly with an affected entity have to be passive to reach the safe state, even small reconfigurations involving a few nodes result in substantial disruptions to the system.
- The re-establishment of application invariants is done through routines embedded in nodes.

To address the above problems, new approaches driving the system to consistency preserving state are studied in [Gou99]. An important assumption for these new approaches is that a component can initiate a transaction unilaterally or as a result of receiving a request. Components do not serve more than one transaction at a time. This is to say that they will not initiate a new transaction or serve any other requests until the transaction they are currently engaged in terminates. Thus, it is possible to drive a component to a quiescent state by blocking its execution when no transactions are being served. Nevertheless, the class of distributed systems to which this alternative can be applied is much more limited than in the case of Kramer and Magee, since components in this approach cannot treat more than one transaction simultaneously.

The simplest way to drive a system into a safe state for reconfiguration is to enforce the blocked state on all the components in the system: A coordinator node sends a "block" message to all the components in the system. As components block they send an "acknowledge" message back to the coordinator. Once all the components in the system have sent the acknowledge message we can be sure that all the components in the system are blocked. The strategy of the algorithm is to prevent new transactions from starting, and also to allow already started transactions to terminate. The drawback of the basic algorithm is obvious: all components in the system are blocked. An improved approach is further proposed by Goudarzi to enforce the blocking only on components that are directly affected by the reconfiguration.

Warren

Warren and his co-authors proposed a dynamic reconfiguration model and reconfiguration algorithms in [WS96, War00]. The algorithm works as follows. Firstly, any component that has a communication path directed to the component to be changed is blocked, causing any requests issued to be queued. The component to be changed

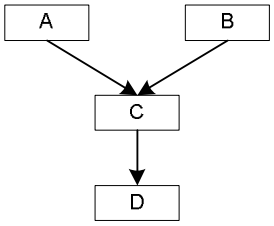

Figure 2.5: *component reconfiguration studied by Warren*

is blocked using an abort block, which prevents this component from initiating new requests of other components. All that remains is to wait for interactions that were already in progress to complete. Once this happens, the component can be safely replaced. Once the change is made, the blocks on the connected components are released and the queued requests are sent to the changed component. For example, let's consider a system configuration shown in Figure 2.5, where the three components communicate asynchronously. To replace component "C", component "A" and component "B" would both be blocked with a queueing block and component "C" would be blocked with an abort block. After waiting for component "C" to finish processing ongoing requests that may have been received from component "A" and component "B", component "C" can be replaced. component "A" and component "B" would then be unblocked and the system would continue functioning with the new component in place.

Wegdam

The development on component-based software engineering, e.g., CORBA [AVSN01], enterprise JavaBean [RAC+02], have enabled efficient implementation of dynamic reconfiguration systems. In [Weg03], a dynamic reconfiguration mechanism is proposed for component-based middleware. This mechanism considers reconfigurable software components that can be manipulated through a set of reconfiguration operations, i.e., creation, replacement, migration and removal. Wegdam presented the mechanism according to the three reconfiguration correctness requirements explained earlier.

- Structural integrity

To preserve interface compatibility after a reconfiguration, a new version of a component must satisfy the original interfaces. Based on Liskov's substitution principle [Lis87], in order to satisfy the original interfaces, the new interfaces have to imple-

ment the original interfaces, or implement interfaces that are subtypes of the original interfaces.

- Mutually consistent states

Wegdam proposed a mechanism to drive the system to the safe state that uses information obtained from the middleware platform at run-time and freezes system interactions on-demand. This mechanism consists of three stages:

1. Drive the system to the safe state by deferring invocations that would prevent the system from reaching the safe state.
2. Detect that the safe state has been reached.
3. Apply reconfiguration.

The emphasis of this mechanism is on the first stage, i.e., reaching the safe state. Wegdam defines the term *affected component* to denote a reconfigurable component that is replaced, migrated or removed as a consequence of reconfiguration. The system is said to be in the reconfiguration safe state when every affected component is not currently involved in invocations and will not be involved in invocations until after reconfiguration. Depending on the way of handling invocations, components are divided into *active* and *reactive* components. Reactive components only initiate requests that are causally related to incoming requests. Active components can initiate requests that do not depend on incoming requests, e.g., they may initiate requests as a result of the elapsing of a time-out. In order to reach the safe state, the main tricks of Wegdam's mechanism are (1) to isolate all the affected components by selectively queueing up requests addressing to them and (2) to force all active affected components exhibit reactive behavior.

- Application state invariants

Each reconfigurable component must provide operations to inspect and modify its state. It is the responsibility of component developers to decide on which kind of state information should be exposed by these state-access operations. In general, a component should provide operations to inspect and modify its control and data state. These operations are only invoked in the safe state.

2.3.3 Stateless Mechanisms

Mitchell

Mitchell [Mit00] describes an algorithm for dynamic replacing some components in a data processing chain (pipe and filter architectural style). This algorithm essentially functions by running partial chains in parallel and switching between these

chains at appropriate time to maintain a consistent flow of data emerging from the last component in the chain. For example in a chain of components “A”, “B”, “C” connecting in sequence, to replace “B” in the chain with “N”, the first step would be to create the replacement component “N”. “A” would then be disconnected from “B” and then connected to “N” so that it can begin processing the output of “A”. Then, once “C” exhausts the output from “B”, it’s input is switched to the output of “N”, resulting in the new chain shown. However, this algorithm has one important applicability constraint which is that the system it reconfigures must conform to the pipe and filter architectural style. Additionally, it requires that components in the system be stateless, as it carries out reconfiguration actions with no concern for state transfer and that components communicate asynchronously. Whilst the case where components are stateless is a rare one, this algorithm serves as an example of an optimization for a particular domain or application type.

Zhao and Li

In [ZL07], the dynamic reconfiguration is studied based on a stateless RDF (Reconfigurable Data Flow) model, which is an extension to the conceptual Data Flow model [JHM04]. The fundamental elements of the RDF model are *process*, *data-store*, and *data-path*. A process is a software component that takes data through its input ports and produces data through its output ports. A data-store is a random-accessible data container which can buffer the transmitted data flow. A data-path is a connector between a process and a data-store through which data can flow. The elementary reconfiguration operations of the RDF model include addition and removal of processes, data-stores or data-paths. In a restricted segment formed by affected elements, the first process is the *entrance process*, the last one is the *exit process*, and between them are *in-between processes*. A coordinator entity plays a central role in the dynamic reconfiguration: it receives the specification of new configuration from the administrator, calculates the role of every process, and coordinates the reconfiguration steps. The proposed dynamic reconfiguration involves five rounds of propagating the control messages through the restricted segment to set up a new route, remove the old route and redirect the data flows.

2.3.4 Concluding Remarks

In the dynamic reconfiguration of MPMS, 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 thesis. In other words, we aim at a stateful mechanism for dynamic task distribution.

Chapter 3
Support Dynamic Task Distribution

This chapter presents the design overview of our proposed adaptation mechanism for MPMSs based on a dynamic task distribution approach. Section 3.1 studies several concepts in system performance and summarizes a set of requirements of this adaptation mechanism. Section 3.2 proposes a new middleware to support dynamic task distribution in an MPMS. Section 3.3 formulates the problem of task assignment in an MPMS.

3.1 Needs for Adaptation

In this section, we discuss the cause of adaptation and the requirements of suitable task assignments in an MPMS. In particular, the first research question (RQ1) raised in Section 1.3 will be answered.

3.1.1 Instant Mismatch vs. Potential Mismatch

A mismatch between task demands and resource supplies is the initial cause for system adaptation [Sat04]. We distinguish between two kinds of mismatch: *instant mismatch* and *potential mismatch*. An instant mismatch occurs when any required task, *currently*, cannot be supported by its associated system resource. All software programs require some features to be present on a computer system before it can be used by the computer. For example, in the DMTF standards[1], this requirement is termed as to "pass the checks". These features include both software and hardware resources. The examples of software resources are the operating system, certain library files, etc. The examples of hardware resources are CPU, Memory, special peripheral devices, present communication channels, etc. An instant mismatch is often caused by an unexpected change in those resources, e.g. a user roamed from an area with a Wireless Local Area Network (WLAN) coverage to a GPRS only area.

A potential mismatch is identified when any required task, *in a foreseeable future*, cannot be supported by its associated system resource. For example, removal of one

[1] http://www.dmtf.org/standards/cim/cim_schema_v214

library file in the future will disable the operation of some dependent programs, or a particular hardware part will shut down in a few minutes due to the quick draining of battery power. For the purpose of "looking into the future", some input or prediction about user's future behavior [WvHK06, GHB08] is often required in order to identify a potential mismatch.

3.1.2 Performance Assurance and Optimization

From the perspective of the end user of an MPMS, e.g., the doctor (Figure 1.1), the following performance measures are influential to the success of the m-health mission. This listing does not mean to cover all important performance characteristics of an MPMS. For example, data security and privacy are also critical factors but not the main concern in this thesis.

1. "End-to-end delay ": defined as the elapsed time between an MPMS receiving an unit of patient's bio-signal information and sending the corresponding processed result of this particular data unit to the decision point. This parameter indicates how quickly the bio-signal and processed result can be delivered by the MPMS. The faster the processed result is delivered, the higher the chance that the patient's emergency situation can be dealt with in time, e.g. in the time critical epilepsy detection application (Section 1.1.2).

2. "System battery lifetime ": defined as the minimum battery lifetime of all the battery powered devices in the system. An MPMS consists of multiple battery powered devices. If the remaining battery energy of a device is lower than a threshold level, it cannot perform bio-signal processing operations any more and the entire MPMS will fail. This parameter indicates the maximum operating time of the MPMS.

3. "Availability level ": consider here to be the steady state availability as defined in [MFT00], that is the application mean uptime divided by the sum of the mean uptime and mean downtime. We assume that failures may potentially occur during either data processing or communication.

Besides the required match between each individual task and its hosting system resource, a user may also have his own specific performance requirements over different measures when the system is treated as a whole. Some of these are (possibly joint) assurance requirements, e.g. "the end-to-end delay should be smaller than 2 seconds" or "the availability level should be higher than 97%". Some are optimization requirements with a single objective, e.g. "the system battery lifetime should be maximized". Some are the combinations of previous two, e.g. "the system battery

lifetime should be maximized while its end-to-end delay should be smaller than 3 seconds"

3.1.3 Suitable Task Assignments

The core of the studied adaptation mechanism is a decision-making component. This component can derive a new task assignment that is more suitable under the new situation. Based on the above discussions, we summarize the requirements of suitable task assignments into four levels ranging from elementary to advanced. They are:

- Level 1: The task assignment should not cause any instant mismatch, i.e., an MPMS configured according to this task assignment should be able to operate in the present situation.
- Level 2: Including the level 1 requirement, the task assignment should not cause any potential mismatch, i.e., an MPMS configured according to this task assignment should be able to operate in the present situation and can still operate in a foreseeable future. To have a completely correct prediction about future situation is never possible, hence, the requirement on operating in a foreseeable future should be interpreted as doing so with a sufficiently high possibility.
- Level 3: Including the level 2 requirement, the task assignment should result in an MPMS that satisfies all performance assurance requirements.
- Level 4: Including the level 3 requirement, the task assignment should result in an MPMS that satisfies the performance optimization requirement.

We admit that it is not always possible for an MPMS to find a suitable task assignment given its particular context and the user defined requirements. Under this circumstance, other system adaptation means have to be applied, e.g., to inform the user about the mismatch situation.

3.2 MADE - A Middleware Level Solution

In an m-health platform, there exists multiple devices with different hardware architecture and software operating system. Hence, the studied adaptation mechanism should work in such heterogeneous environment. A middleware level solution is introduced in this section. Middleware refers to a distributed platform of interfaces and services that reside "between" the application and the operating system and

aim to facilitate the development, deployment and management of distributed applications [Cou00]. The main function of middleware is to hide the heterogeneity of distributed systems, e.g. communication networks, different programming languages and operating systems, and provide a standard set of interfaces and services which distributed applications can assume present.

3.2.1 MADE Overview

The main goal of the studied adaptation mechanism is to adapt the configuration of an MPMS. Thus, as illustrated in the scenario in Section 1.1.2, an MPMS can respond to context changes by properly redistributing the bio-signal processing tasks at runtime. To achieve this, we propose a middleware providing four main functionalities: *Monitoring*, *Analysis*, *Decision* and *Enforcement* (MADE) shown in Figure 3.1. The monitoring phase includes registration of the telemonitoring application, device discovery, resource monitoring, and context discovery/registration. The analysis phase takes the information about an MPMS as input and runs a task assignment algorithm to provide suitable task assignments (candidate assignments) that can satisfy user's requirements. The task assignment algorithm can be executed based on a predetermined schedule or triggered by a "significant" context change, e.g. a user moves out of an area of WLAN coverage into a GPRS only area. The decision phase compares the candidate assignments with the current system configuration to determine the actual cost of reconfiguration. If the reconfiguration cost can be covered by the enhanced performance of the new configuration, a new target assignment plan will be identified and executed. The Enforcement phase controls the MPMS to adjust its configuration according to the new target assignment.

3.2.2 Architectural Components

The system architecture of the Monitoring, Analysis, Decision, Enforcement (MADE) middleware is depicted in Figure 3.2. In this architecture, we identify two kinds of software components, i.e., a Coordinator and a set of Facilitators. These software components form an overlay network on top of the m-health platform. Each device that is part of the m-health platform should deploy locally a "Facilitator" which in turn represents its hosting device in MADE: it monitors and reports its local context information (about the hosting device) to the Coordinator and receives control commands from the Coordinator to dynamically reconfigure its hosted tasks. The core of Coordinator is a task assignment algorithm that computes the suitable task assignments. This computation is performed based on the required telemonitoring application and the current context information about the m-health platform,

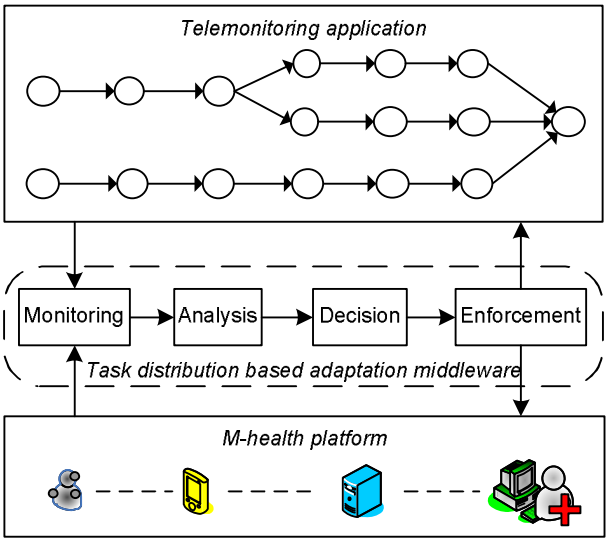

Figure 3.1: *MADE - a middleware layer that support task distribution based adaptation*

e.g. available devices and their connectivity, each device's CPU load and remaining battery energy, etc. After a target task assignment is identified, the Coordinator translates the difference between the current system configuration and the new target task assignment into a reconfiguration plan. Then the Coordinator controls the hosting Facilitators to deploy the target task assignment by means of task distribution.

3.2.3 Behavior

After presenting the high-level system architecture, we explain here how MADE should support dynamic task distribution. We do so by illustrating its runtime behavior as depicted in Figure 3.3. The initial step of MADE is "context monitoring". In this step, MADE monitors resource usage and context changes in the m-health platform and can be informed about a significant performance variation occurred in the telemonitoring application. Once a mismatch (between application demand and resource supply) is detected, MADE moves to a new step where it can compute a set of new candidate assignments. These new candidate assignments should satisfy the "level 3" requirement defined in Section 3.1. In the next step, MADE identifies the new target assignment from this set of candidate assignments. This decision is made by taking into consideration of both the performance enhancement and the cost of

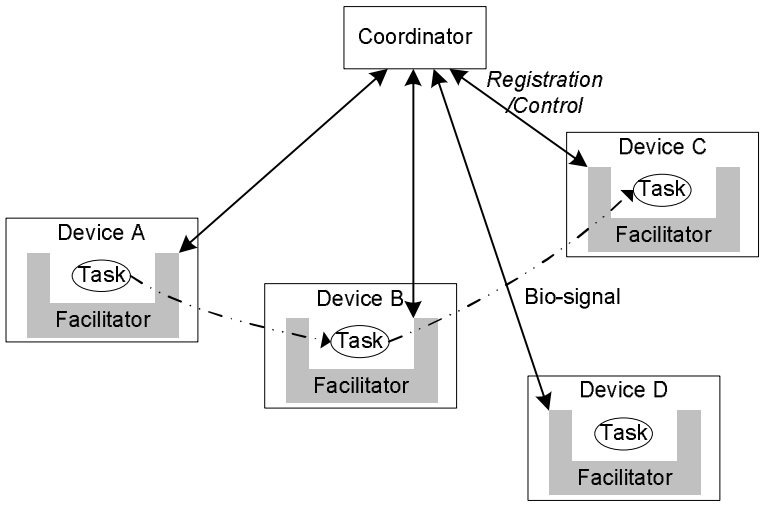

Figure 3.2: *The system architecture of MADE middleware*

reconfiguration. We will elaborate more on these "Analysis" and "Decision" behaviors in Section 3.3.3, in Chapter 4 and Chapter 5. The discussion on reconfiguration cost takes place in Chapter 6.

It is possible that in some situation the new target assignment just does not exist, i.e., there is no single task assignment can satisfy all the requirements. If this happens, we can conclude that the MPMS can not be adapted by redistributing tasks to survive the new situation. In this case, MADE should move to a step where users can be informed and other adaptation mechanisms can be called if there are any.

If a new target assignment is successfully determined, MADE will move to the steps of "Calculate a reconfiguration plan" and "Reconfigure the MPMS" subsequently. We will elaborate more on these "Enforcement" behaviors in Chapter 6.

Upon completing a successful reconfiguration, MADE moves back to the initial "context monitoring" step and waits until next mismatch is observed. If the reconfiguration fails due to some unexpected errors, e.g., an involved device suddenly becomes unavailable, the on-going reconfiguration has to be terminated and the system has to be rollback to the last working configuration. If the rollback is done without compromising the system consistency, MADE should move back to the initial "context monitoring" step and may start calculating new task assignments again with new context information. If the rollback fails, the MPMS may suffer a potential system consistency and jeopardize the supported telemonitoring application. Thus, we should alert users with this situation and let it be handled properly.

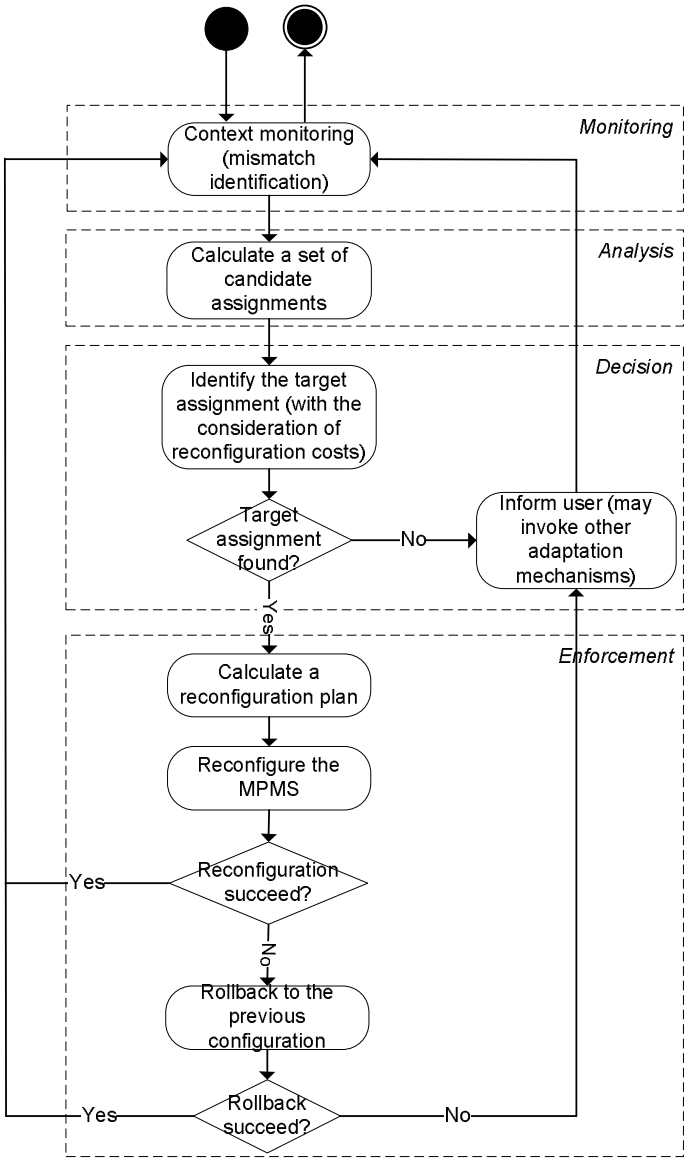

Figure 3.3: *The behavior overview of MADE middleware*

3.3 Decision-Making Core of MADE

The core of the MADE middleware layer is the "analysis" phase where a task assignment algorithm is required to compute the optimal task assignment given the application/system context information. In this section, we discuss this task assignment problem in more detail and present the mathematical model of the problem.

3.3.1 Definitions

We model a telemonitoring application as a partial order of bio-signal streaming tasks. We distinguish two types of streaming tasks: stream processing tasks and stream transmission tasks. Processing tasks typically perform some operation on the bio-signal stream such as filtering, transcoding, or other m-health relevant data processing operations. Each processing task consumes one or more data streams and produces one or more data streams. Transmission tasks are the glue between processing tasks and have two functions: firstly they allow us to easily characterize properties of the data stream (for instance the data rate of the stream); and secondly, as we will see later, transmission tasks can be mapped onto a communication path (such a path may be a stream pipe within a device, or it may be a networked path between different devices) and hence support the modeling of relaying devices.

A telemonitoring application consisting of distributed tasks can be defined as a tuple of (P, T, A_t, L_P, L_T), where P is a set of stream processing tasks $\{p_1, p_2, ...\}$, T is a set of transmission tasks $\{t_1, t_2, ...\}$, A_t is a set of precedence relations between tasks, such that $A_t \subseteq P \times T \cup T \times P$, L_P is a set of labels over each processing task, L_T is a set of labels over each transmission task. L_P and L_T indicate the resource demand of processing tasks and transmission tasks respectively, a detailed overview is presented in Table 3.1. The structure $\{P, T, A_t\}$ is a DAG termed as a *task DAG*. An example of a task DAG is presented in Figure 3.4.

Similarly, an m-health platform is defined as a tuple of (D, C, A_r, L_D, L_C), where D is a set of device resources $\{d_1, d_2, ...\}$, C is a set of (communication) channel resources $\{c_1, c_2, ...\}$, A_r is a set of precedence relations between resources, such that $A_r \subseteq D \times C \cup C \times D$, L_D is a set of labels over each device resource, L_C is a set of labels over each channel resource. L_D and L_C model the resource supply of devices and channels respectively, further details are given in Table 3.1. The structure $\{D, C, A_r\}$ is a DAG termed as a *resource DAG*, an example of such a graph is shown in Figure 3.4.

We assume that device resources in an m-health platform can relay bio-signal data streams, therefore a transmission task may be assigned to a directed communication path. A communication path is a directed path in the resource DAG starting

at d_i and ending at d_j. In general, there exist multiple paths connecting d_i to d_j. When $i = j$, it is a special communication path within device d_i and can be denoted the same as the device. We define CP as the set of all communication paths in a resource DAG.

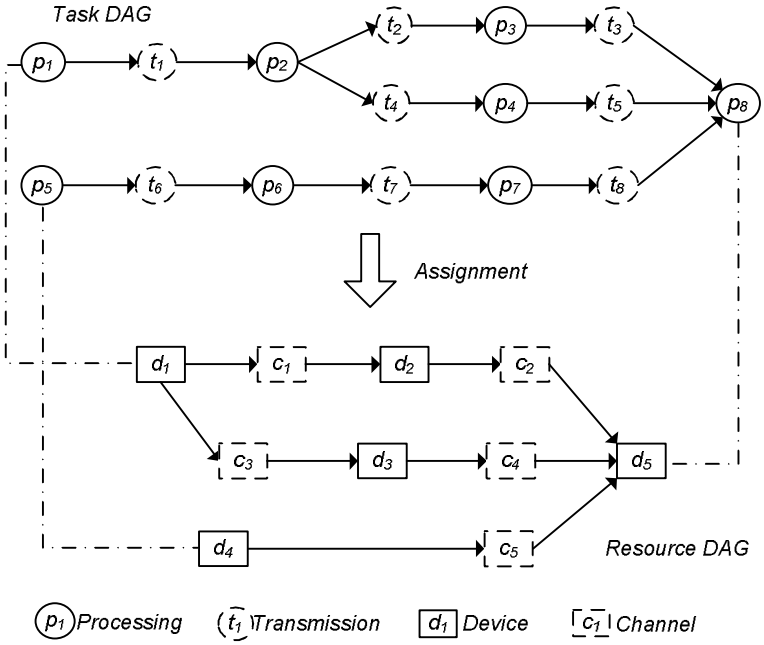

Figure 3.4: *Model of task assignment in an MPMS. Due to the fixed association of sensory tasks with sensors, "p_1" has to be assigned to "d_1" and "p_5" has to be assigned to "d_5".*

Both task DAG and resource DAG are bipartite graphs in which each type of vertex forms a disjoint set, i.e., there is no connection between two vertices of the same type. In a task DAG, each transmission task has exactly one predecessor processing task and one successor processing task, while a processing task may have multiple predecessor transmission tasks and multiple successor transmission tasks. A similar property holds for a resource DAG. Regarding the semantics of the task DAG, we have one important assumption: whenever a processing task has multiple predecessor transmission tasks, the processing task can only function when the bio-signals from all these predecessor transmission tasks are received and synchronized. In effect, we only allow bio-signal forking and joining with AND semantics (as known from workflow theory).

Notation	Belongs to	Meaning
$n^{\mathrm{P}}_{p_i}$	L_P	the number of operations per time unit at processing task p_i
rr_{p_i}	L_P	the required resource of processing task p_i, e.g. minimum required CPU, minimum required memory, etc
$de^{\mathrm{P}}_{p_i,d_j}$	L_P	the processing delay of processing task p_i to process one data unit at device d_j
av_{p_i,d_j}	L_P	the availability of running processing task p_i at device d_j
$n^{\mathrm{T}}_{t_i}$	L_T	the number of transmitted data units per time unit of transmission task t_i
$de^{\mathrm{T}}_{t_i,d_j}$	L_T	the transmission delay of transmission task t_i to transfer one data unit at device d_j
$de^{\mathrm{T}}_{t_i,c_j}$	L_T	the transmission delay of transmission task t_i to transfer one data unit at channel c_j
av_{t_i,c_j}	L_T	the availability of performing transmission task t_i at channel c_j
av_{t_i,d_j}	L_T	the availability of performing transmission task t_i at device d_j
$e^{\mathrm{TO}}_{d_i}$	L_D	the total available battery energy at device d_i[2]
$e^{\mathrm{HK}}_{d_i}$	L_D	the energy consumption rate of the "housekeeping" activities at device d_i, e.g. display, network interface cards, etc
$e^{\mathrm{op}}_{d_i}$	L_D	the energy consumption of one operation at device d_i
rs_{d_i}	L_D	the available resource supply at device d_i , e.g. CPU type, available memory, etc
bw_{c_i}	L_C	the available bandwidth at channel c_i
lo_{c_i}	L_C	the current load information (influenced by other users in the same channel) at channel c_i
$e^{\mathrm{S}}_{c_i}$	L_C	the energy consumption of sending one data unit through channel c_i
$e^{\mathrm{R}}_{c_i}$	L_C	the energy consumption of receiving one data unit through channel c_i

Table 3.1: *Notation for labeling*

Notation	Meaning
P	a set of processing tasks
T	a set of transmission tasks
D	a set of device resources
C	a set of (communication) channel resources
TP	a set of task paths $\{tp_i\}$ in the task DAG
CP	a set of communication paths $\{cp_i\}$ in the resource DAG
A_t	a set of precedence relations between tasks
A_r	a set of precedence relations between resources
Ω^{de}	the measure of end-to-end delay
Ω^{av}	the measure of availability
Ω^{li}	the measure of battery lifetime
$max\Omega^{\mathrm{de}}$	the maximal allowed system end-to-end delay
$min\Omega^{\mathrm{av}}$	the minimal required system availability
$min\Omega^{\mathrm{li}}$	the minimal required system battery lifetime
N_{rc}	the number of required candidate task assignments
$de^{\mathrm{T}}_{t_i,cp_j}$	the transmission delay of transmission task t_i to transfer one data unit through communication path cp_j
$e^{\mathrm{P}}_{d_i}$	the energy consumption rate of data processing at device d_i
$e^{\mathrm{S}}_{d_i}$	the energy consumption rate of sending data stream at device d_i
$e^{\mathrm{R}}_{d_i}$	the energy consumption rate of receiving data stream at device d_i

Table 3.2: *List of Symbols (exclude labeling symbols)*

Based on the two graph models, a task assignment (Φ) is a mapping of tasks onto resources such that: each processing task is mapped to one device and each transmission task is mapped to one communication path:

$$\Phi : P \bigcup T \rightarrow D \bigcup CP \tag{3.1}$$

where $CP = \{x_1, x_2, ..., x_n\}$ is a sequence, $x_1 \in D$, $x_n \in D$, and $\forall 1 \leq i < n :< x_i, x_{i+1} >\in A_r$.

3.3.2 Performance Estimation for A Given Task Assignment

In this section, we present the computational model to estimate system performance, i.e., end-to-end delay, system battery lifetime and availability, given a particular assignment.

End-to-End Delay

In a task DAG, we define task path as a directed path connecting two tasks. The entire set of task paths is denoted as TP. Every task path, tp_i, is an ordered sequence of processing tasks and transmission tasks, and the m^{th} task can be denoted as $tp_i(m)$. Upon different task assignment Φ, tp_i exhibits a different end-to-end delay, i.e., the summation of processing delay and transmission delay along the path:

$$\begin{aligned}\Omega^{\text{de}}(tp_i) \quad = \quad & \sum_{tp_i(m)\in P} de^{\text{P}}_{tp_i(m),\Phi(tp_i(m))} + \\ & \sum_{tp_i(m)\in T} de^{\text{T}}_{tp_i(m),\Phi(tp_i(m))}\end{aligned} \tag{3.2}$$

where $de^{\text{P}}_{p_i,d_j}$ is defined as the processing delay for processing task p_i to process one unit of biosignal data at device d_j, $de^{\text{T}}_{t_i,cp_j}$ is defined as the transmission delay of transmission task t_i to transfer one unit of biosignal data over a communication path cp_j. $de^{\text{T}}_{t_i,cp_j}$ can be computed by adding up the transmission delay occurred at device d_j ($de^{\text{T}}_{t_i,d_j}$), and at channel c_j ($de^{\text{T}}_{t_i,c_j}$). The values of $de^{\text{P}}_{p_i,d_j}$, $de^{\text{T}}_{t_i,d_j}$ and $de^{\text{T}}_{t_i,c_j}$ can be estimated from the profiling information, e.g. $n^{\text{P}}_{p_i}$, $n^{\text{T}}_{t_i}$, bw_{c_i} and lo_{c_i} based on the knowledge from real measurements, e.g. in [XRC+02].

The end-to-end delay of a given assignment Φ, $\Omega^{\text{de}}(\Phi)$, is then defined as the maximum of all task paths' end-to-end delays:

$$\Omega^{\text{de}}(\Phi) = \max(\{\Omega^{\text{de}}(tp_i)|tp_i \in TP\}) \tag{3.3}$$

Battery Lifetime

Based on the power consumption model of a mobile device [RZ07], we estimate the battery life time for a specific device d_i, $\Omega^{\text{li}}(d_i)$, as:

$$\Omega^{\text{li}}(d_i) = \frac{e^{\text{TO}}_{d_i}}{e^{\text{HK}}_{d_i} + e^{\text{P}}_{d_i} + e^{\text{R}}_{d_i} + e^{\text{S}}_{d_i}} \tag{3.4}$$

Where $e^{\text{TO}}_{d_i}$ is the total available battery energy at device d_i, $e^{\text{HK}}_{d_i}$ is energy consumption rate of device's "housekeeping" activities, e.g. display and network interface

cards; $e_{d_i}^{\mathrm{P}}$ is the energy consumption rate by local data processing; $e_{d_i}^{\mathrm{R}}$ is the energy consumption rate for receiving data stream; $e_{d_i}^{\mathrm{S}}$ is the energy consumption rate for sending data stream. Given a particular task assignment Φ, the latter three parameters can be calculated as:

$$e_{d_i}^{\mathrm{P}} = e_{d_i}^{\mathrm{op}} \sum_{\{p_j|\Phi(p_j)=d_i\}} n_{p_j}^{\mathrm{P}} \tag{3.5}$$

$$e_{d_i}^{\mathrm{S}} = \sum_{(d_i,c_j)\in A_r} (e_{c_j}^{\mathrm{S}} \sum_{c_j\in\Phi(t_k)} n_{t_k}^{\mathrm{T}}) \tag{3.6}$$

$$e_{d_i}^{\mathrm{R}} = \sum_{(c_j,d_i)\in A_r} (e_{c_j}^{\mathrm{R}} \sum_{c_j\in\Phi(t_k)} n_{t_k}^{\mathrm{T}}) \tag{3.7}$$

Once the battery lifetime of all devices are estimated, the minimum of all device's battery lifetime determines the overall system lifetime for a given assignment:

$$\Omega^{\mathrm{li}}(\Phi) = \min(\{\Omega^{\mathrm{li}}(d_i)|d_i \in D\}) \tag{3.8}$$

Availability

Since we assume only AND semantics in task DAG, the availability of the application depends on all the included tasks: The application performs successfully only when all tasks perform successfully. Therefore, the application availability level for a given assignment Φ, $\Omega^{\mathrm{av}}(\Phi)$, can be computed as:

$$\Omega^{\mathrm{av}}(\Phi) = \prod_{\mathrm{p_i}\in\mathrm{P}} \mathrm{av}_{\mathrm{p_i},\Phi(\mathrm{p_i})} \cdot \prod_{\mathrm{t_i}\in\mathrm{T},\alpha\in\Phi(\mathrm{t_i})} \mathrm{av}_{\mathrm{t_i},\alpha} \tag{3.9}$$

3.3.3 Problem Formulation

We formulate the task assignment problem in MPMS as follows:

- **Given**: (1) a telemonitoring application (P, T, A_t, L_P, L_T); (2) an m-health platform (D, C, A_r, L_D, L_C); (3) Three performance evaluation function Ω^{li}, Ω^{de} and Ω^{av} for task assignments.
- **Goal**: To find a number of candidate assignments $\{\Phi_m^{\mathrm{can}}|m = 1, 2, N_{rc}\}$ among all possible task assignments such that the value of $\Omega(\Phi_m^{\mathrm{can}})$ are optimized. One special case is that when $N_{rc} = 1$. Thus, the goal is to find the optimal task assignment Φ_{opt}.
- **Subject to**: four assignment constraints namely *type constraint*, *local constraint*, *assurance constraint* and *reachability constraint*.

The *type constraint* specifies that each processing task must be mapped to one and only one device resource and each transmission task must be mapped to one and only one communication path, hence:

$$\forall p_i \in P : \Phi(p_i) \in D, \forall t_i \in T : \Phi(t_i) \in CP \tag{3.10}$$

The *local constraint* comprises two parts:

1. For every processing task p_i in P, its assigned device must be able to provide the task's required resources. In MPMSs, this is often the case when a sensing task has to be associated with a device with a supporting sensor, e.g. the fixed associations illustrated in Figure 3.4. That is, assuming we have a Boolean function $satisfy(rr, rs)$ to evaluate whether a required resource, rr, can be satisfied by a resource supply, rs. Thus:

$$\forall p_i \in P : satisfy(rr_{p_i}, rs_{\Phi(p_i)}) = true \tag{3.11}$$

2. For each channel resource, the total assigned transmission tasks must not exceed its offered bandwidth:

$$\forall c_i \in C : \sum_{c_i \in \Phi(t_i)} n^{\mathrm{T}}_{t_i} < bw_{c_i} \tag{3.12}$$

For an MPMS, we denote the minimum required availability as $min\Omega^{\mathrm{av}}$, the minimum required system battery lifetime as $min\Omega^{\mathrm{li}}$, and the maximum allowed system end-to-end delay as $max\Omega^{\mathrm{de}}$. Thus the *assurance constraint* for selecting candidate task assignments is defined as:

$$\Omega^{\mathrm{de}}(\Phi^{\mathrm{can}}_m) < max\Omega^{\mathrm{de}}, \Omega^{\mathrm{av}}(\Phi^{\mathrm{can}}_m) > min\Omega^{\mathrm{av}}, \Omega^{\mathrm{li}}(\Phi^{\mathrm{can}}_m) > min\Omega^{\mathrm{li}} \tag{3.13}$$

The *reachability constraint* specifies that for each processing task p_i assigned to device resource $\Phi(p_i)$, its predecessor transmission task must be assigned to a communication path ending at $\Phi(p_i)$, and its successor transmission task must be assigned to communication path starting at $\Phi(p_i)$.

3.3.4 Towards A General Solution

For some MPMSs, the task assignment problem can be modeled with reduced complexity. In particular, we studied the general "DAG-DAG" task assignment problem with three specializations: (1) "DAG-tree" where the telemonitoring application is a DAG and the m-health platform has a tree structure; (2) "chain-chain" where both the monitoring application and the m-health platform exhibit as a chain; (3)

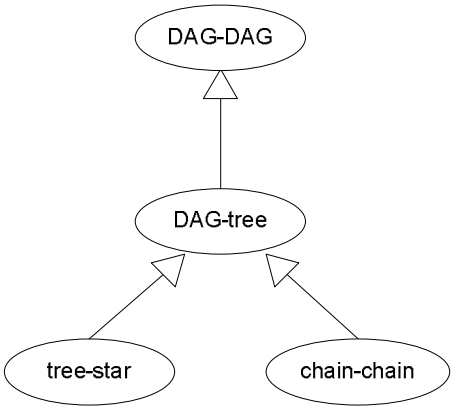

Figure 3.5: *Taxonomy of task assignment problems*

"tree-star" where the telemonitoring application has a tree structure and m-health platform has a star structure. Subtype-supertype relations exist among these four types as shown in Figure 3.5, e.g. "DAG-DAG" is a supertype of "DAG-tree". The solutions available for the supertype problem can be also applied to the subtype problem. However, due to the specialization of the subtype problem, there may exist more efficient algorithms to compute solutions for the subtype problem. In Chapter 4, two of these specializations are studied: "chain-chain" and "tree-star". For readers who are interested in the problem of "DAG-tree", we refer them to [MvBBW+09]. The general form of the problem, i.e., "DAG-DAG", is discussed in Chapter 5.

Chapter 4

Task Assignment Algorithms in Special Topologies

The general model of task assignment problem in an MPMS has been formulated in the previous chapter. Additional flexibility is achieved in task assignments by allowing transmission tasks to be assigned to a communication path, hence, relaying devices can be supported by our model. In its general form, task assignment is a well known NP-hard problem. However, as some earlier work have proven (c.f. Section 2.2.3), in case the task model and resource model obey certain topology and the cost model satisfies certain property, dynamic programming methods can be applied and polynomial-time algorithms may exist to solve these specific problems.

In this chapter, we study two specific forms of task assignment problems, namely "chain-chain assignment" and "tree-star assignment". Both forms have been studied earlier by various researchers (c.f. Section 2.2.3). Our study differs from those earlier work in the following three aspects:

1. Our resource model permits the existence of relaying devices.

2. Our cost model is different, i.e., minimizing the end-to-end delay is used as the objective function.

3. A number of assignment constraints are relaxed or modified to reflect the specific features of MPMSs.

4.1 Chain-Chain Assignment - Exact Algorithm

When both the telemonitoring application and the m-health platform form a chain as illustrated in Figure 4.1 and the goal of the task assignment is to find an optimal assignment with minimal end-to-end delay, polynomial-time task assignment algorithm exists. For example, given the task assignment shown in Figure 4.1, the end-to-end delay can be computed as follows based on the labeling defined in Ta-

ble 3.1:

$$\begin{aligned}\Omega^{\mathrm{de}} &= de^{\mathrm{P}}_{p_1,d_1} + de^{\mathrm{P}}_{p_2,d_1} + de^{\mathrm{P}}_{p_3,d_3} + de^{\mathrm{P}}_{p_4,d_3} + de^{\mathrm{P}}_{p_5,d_3} + de^{\mathrm{P}}_{p_6,d_4} \\ &\quad + de^{\mathrm{T}}_{t_1,d_1} + de^{\mathrm{T}}_{t_2,c_1} + de^{\mathrm{T}}_{t_2,d_2} + de^{\mathrm{T}}_{t_2,c_2} \\ &\quad + de^{\mathrm{T}}_{t_3,d_3} + de^{\mathrm{T}}_{t_4,d_3} + de^{\mathrm{T}}_{t_5,c_3} \end{aligned} \tag{4.1}$$

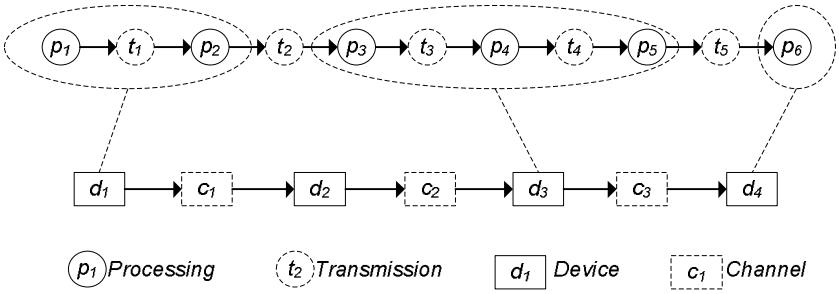

Figure 4.1: *An example of assigning a directed task chain containing 6 tasks onto a directed resource chain containing 4 devices*

4.1.1 Formulation

In a chain-chain assignment, once two adjacent processing tasks are assigned to device resources, there is only one communication path connecting these two devices to which the transmission task in between can be assigned. This permits a simpler formulation of the chain-chain assignment problem by neglecting transmission tasks and channels that is presented as follows:

- Given: (1) a directed task chain containing m processing tasks ($\{p_i\}$) increasingly indexed by $i \in \{1, 2, \ldots, m\}$; (2) a directed device chain containing n devices ($\{d_j\}$) increasingly indexed by $j \in \{1, 2, \ldots, n\}$; (3) An end-to-end delay evaluation function Ω^{de} for task assignments.
- Goal: To find the optimal task assignment Φ_{opt} among all possible task assignment functions $\{\Phi_1, \Phi_2, ...\}$ such that:

$$\forall i \in \{1, 2, ...\} : \Omega^{\mathrm{de}}(\Phi_{\mathrm{opt}}) \leq \Omega^{\mathrm{de}}(\Phi_i) \tag{4.2}$$

- Subject to: the following reachability constraint and local constraint.

The reachability constraint is that for every pair of task p_i and task $p_{i'}$: if $i < i'$, then $\Phi(i) \leq \Phi(i')$. This constraint therefore reflects the alignment of directions of the two chains.

The local constraint is that for every task p_i, its hosting device $\Phi(i)$ should satisfy the task p_i's system requirement and user preference.

4.1.2 Solution

Given a task chain (of m processing tasks) to a device chain (of n devices) assignment problem, an efficient polynomial-time assignment algorithm is illustrated in this section. We first build a layered assignment graph consisting $mn + 2$ vertices (Figure 4.2). In this graph, each row (excluding vertices "$< S >$" and "$< T >$") corresponds to a processing task and each column corresponds to a device. The label "$< p_i, d_j >$" on each vertex corresponds to a possible (i.e., satisfying the local constraint) assignment of processing task p_i to device d_j. A vertex labeled $< p_i, d_j >$ is connected by directed edges to all vertices $< p_{i+1}, d_j >$, $< p_{i+1}, d_{j+1} > \ldots < p_{i+1}, d_n >$ in the layer below. Directed edges connect from a source vertex "$< S >$" to all vertices in the first layer and connect from all vertices in the last layer to the sink vertex "$< T >$". Therefore, any directed path connecting vertex "$< S >$" to "$< T >$" corresponds to an assignment of processing tasks to devices fulfilling the reachability constraint. For example, the thick path in Figure 4.2 represents the assignment shown in Figure 4.1).

Each edge of this layered assignment graph is then labeled with a weight representing the sum of processing delays and transmission delays such that: (1) the edges connecting vertex "$< S >$" to all vertices in the first layer have a weight 0; (2) For layer i (except for the last layer), each edge connecting from the vertex "$< p_i, d_j >$" to the vertex "$< p_{i+1}, d + k >$" is labeled with a weight equal to the sum of processing delay (the delay for p_i to process one data unit at device resource d_j, i.e $de^{\mathrm{P}}_{p_i,d_j}$) and the transmission delay (the delay caused the transmission over the channels connecting d_i and d_k); (3) Each edge connecting the vertices in the last layer, "$< p_m, d_j >$",to the vertex "$< T >$" is labeled with a weight that equals the processing delay of $de^{\mathrm{P}}_{p_m,d_j}$. As an illustration, the weights associated with the thick edges are shown in Figure 4.2.

In the last step, by applying a shortest-path search algorithm (e.g., Dijkstra's algorithm), we can identify a shortest path connecting "$< S >$" and "$< T >$" in this weighted graph that corresponds to the optimal assignment. The space complexity (defined as the number of vertices in the assignment graph) of this method is $O(mn)$. Thus the time complexity of shortest-path search step is $O(m^2n^2)$ if the Dijkstra algorithm with the simplest implementation is used. If a labeling method as proposed by Bokhari [Bok88] is used for search, the time complexity can be reduced to $O(mn)$.

It is easy to convert our method to tackle the assignment problem with the con-

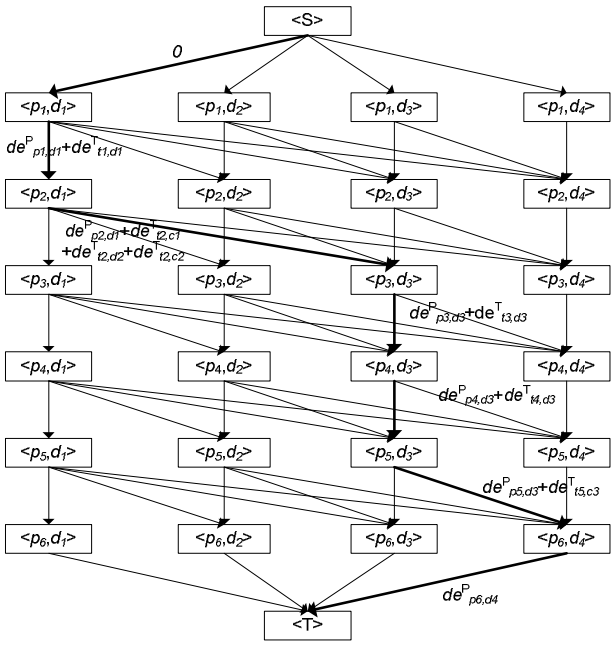

Figure 4.2: *A chain of 6 processing tasks to a chain of 4 devices assignment graph*

tiguity constraint, i.e., the original problem studied by Bokhari [Bok88]. We only need to remove all the edges which are "cutting a vertical edge", i.e., the possible assignment of two adjacent processing tasks onto two non-adjacent devices and all the edges connecting vertices "$< S >$" and "$< T >$" except for the ones connecting "$< p_1, d_1 >$" and "$< p_m, d_n >$". This simplified assignment graph is shown in Figure 4.3. Following the same weighting and search steps as the ones in the assignment problem without contiguity constraint, the optimal assignment with the contiguity constraint can be obtained by searching the shortest path in this new assignment graph. This new assignment graph has some sort of relation with some earlier proposed assignment graphs [SC90, HL92, Yeh05]: the vertex matrix in our solution is the transpose of the vertex matrix proposed earlier that represents the search space of task assignments with the contiguity constraint.

4.1.3 Performance Analysis

We implemented this proposed chain-chain assignment algorithm in Java to get more in-depth understanding. This is, to the best our knowledge, the first experimental study based on series of theoretical studies on chain-chain assignment prob-

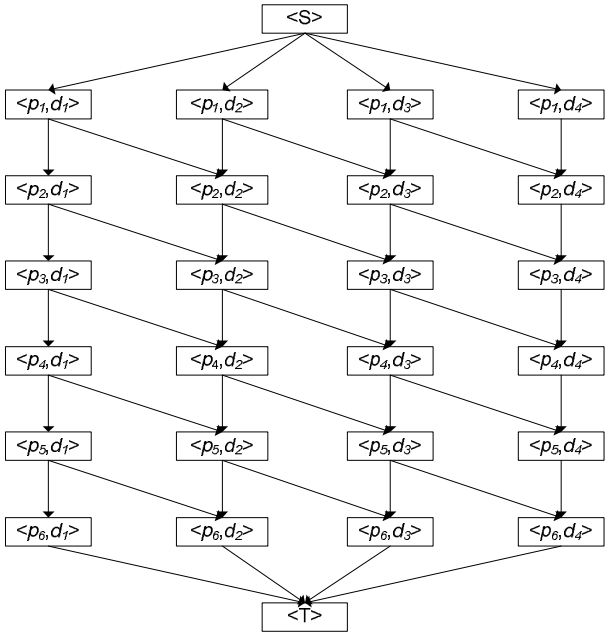

Figure 4.3: *A chain of 6 processing tasks to a chain of 4 devices assignment graph with the contiguity constraint*

lem reviewed earlier (c.f. Section 2.2.3). We recorded the CPU time of two computation steps in this method: the weighted assignment graph construction and the shortest-path search. For the "search" step, we use an open source library [1]. This library implements the Dijkstra shortest-path algorithm with the support of a real priority queue that keeps the order at all time. Thus the time complexity of the search is $O(|E| \log |V|)$, where $|E|$ is the number of edges and $|V|$ is the number of vertices. The program is tested on a Windows XP machine with Intel Pentium 2.4G CPU and 1.5G RAM. Different pairs of m and n, e.g., (20,10), (40,10), ..., (100,20), are tested and the number of assignment graph nodes ranges from 200 to 2000. Both the assignment with the contiguity constraint and the assignment without it are tested (Figure 4.4).

One observation we made is that the "construction" step is more time consuming than the actual "search" step in most cases, which suggests that the evaluation on this kind of graph-based method should be based on the time complexity of the entire method instead of the "search" step only [Bok88, SC90, NO91]. Furthermore, we also implemented a program based on the optimal assignment approach proposed

[1] http://rollerjm.free.fr/pro/graphs.html

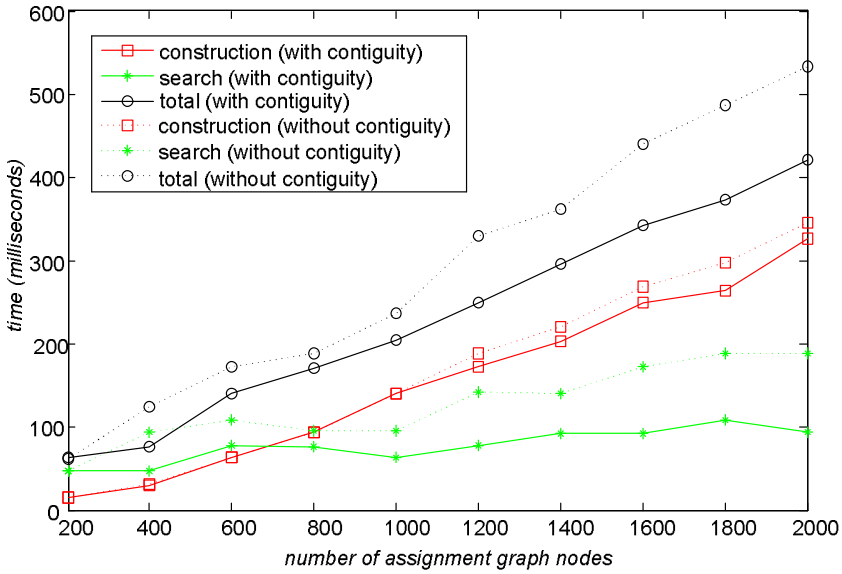

Figure 4.4: *The CPU time of searching for the optimal chain-chain assignment*

by Bokhari [Bok88] and tested against our method with the contiguity constraint. Again, the CPU time of graph construction and shortest-path search are measured separately. Several pairs of m and n are tested and the results are shown in Table 4.1. Due to the reduced time and space complexity, our method outperforms the original Bokhari's method. For an assignment problem of m processing tasks and n devices, our method requires an assignment graph with $mn + 2$ vertices compare to a Bokhari's graph with $O(m^2n)$ vertices[2]. For example, for a "60-task-30-device" assignment problem, an assignment graph with 14952 vertices will be constructed if Bokhari's method is used. Because we use the adjacent matrix to represent the graph, which implies heavy memory requirement, the program runs out of memory before the completion. In our new method, the assignment graph has 1802 vertices and an optimal assignment can be found.

4.2 Chain-Chain Assignment - Genetic Algorithm (GA)

Although the proposed algorithm in the previous section is more efficient compared to earlier work, it still suffer a major drawback on memory consumption. This graph-based algorithm requires an assignment graph to be constructed first which

[2]To be more precise, it is $(m - n + 2) * (m - n + 1) * (n - 2)/2 + (m - n + 1) * 2 + 2$

Setting		CPU time of Bokhari's method			CPU time of the new method		
m	n	Const	Search	Total	Const	Search	Total
20	10	94	63	157	16	47	63
30	10	392	172	564	31	47	78
40	10	985	673	1658	32	46	78
30	20	250	94	344	63	78	141
40	20	1017	563	1570	94	78	172
50	20	3996	2690	6686	141	94	235
40	30	360	156	516	172	94	266
50	30	2112	939	3051	235	109	344
60	30	Out of memory	Out of memory	Out of memory	282	110	392

Table 4.1: *CPU time (in millisecond) comparison between Bokhari's original method and our proposed method on chain-chain assignment with the contiguity constraint*

becomes a bottleneck of the solution (Table 4.1). This motivates us to investigate further on this problem and seek for a solution that is more friendly on memory consumption. In this section, we propose a GA based solution that indeed provides such an advantage. Genetic algorithms have been applied earlier on the task assignment problem [WSRM97, CFR99]. However these GA solutions proposed earlier can not be applied to our task assignment problem in their original form because the different underlying resource model. The resource model in [WSRM97, CFR99] is a fully connected network with symmetric links while we consider a chain-like structure in this section. Furthermore, as compared to [WSRM97, CFR99], our approach benefits from the simpler chromosome encoding (c.f. Section 4.2.2) and less processing intensive crossover and mutation operations.

4.2.1 General Design

GA is a proven heuristic technique based on the "survival of the fittest" paradigm to find the sub-optimal solution in the large search spaces. The first thing to do for applying GA is to encode possible solutions as a set of strings (chromosomes). Each chromosome represents one solution to the problem. There exist several encoding techniques such as binary encoding, number encoding, character encoding and tree encoding. A set of chromosomes is referred to as a population.

The second step is the generation of an initial population. Usually, the chromo-

somes which constitute an initial population are generated purely randomly. However, to take care of the reachability constraint, we introduce certain ordering in the generation of the initial population. For the second step, the initial population represents the current generation. The number of chromosomes constituting the current generation is known as the population size. The population size affects the quality of the sub-optimal solution obtained.

The third step is to evaluate the fitness of every chromosome in the current generation. A specific objective function representing the utility of the chromosome determines the fitness of the chromosome. In our case, the objective function evaluates the end-to-end delay and a smaller delay represents higher fitness.

The fourth step is applying the selection operator to form the next generation. The selection operator allows the GA to take biased decisions favoring the chromosomes having higher fitness value over the chromosomes having the lower fitness value. As a consequence, each chromosome is removed or selected (one or multiple times) based on its fitness value. There exist several selection techniques. We choose the roulette wheel selection technique for our experimentation. In this technique, the probability of the selection of a chromosome in the next is directly proportional to its fitness value. The population size of the next generation typically remains the same as the initial generation. After selecting the next generation, the current generation is replaced by the initial generation.

Selection is followed by applying the crossover operator over the current generation. With some probability (referred to as crossover probability), some pairs of chromosomes (referred to as parent chromosomes) are selected from the current population and some of their corresponding components are exchanged at the randomly chosen crossover point(s) to form two new chromosomes (referred to as children chromosomes). There are also various types of the crossover involving single point crossover and multi-point crossover. We choose the single point crossover in this thesis. The crossover operator eventually leads to combining the best subsequence of one chromosome with that of the other chromosome to evolve better new chromosomes.

After applying the crossover operator, each element of the chromosome in the population is applied a mutation operator with some probability (referred to as mutation probability). The mutation operator transforms a chromosome into another chromosome and this may eventually lead to a better chromosome which could not be obtained just by applying the crossover operator.

Later, the sequence of steps (referred to as one iteration) involving selection, crossover and mutation operators is repeated until a certain stopping criterion is met. A stopping criterion may involve various forms such as running the GA for a certain number of iterations, or when no improvement in the maximum fitness is

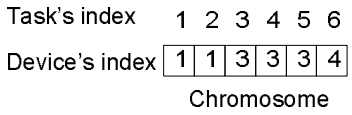

Figure 4.5: *A possible chromosome representing the placement of tasks to the devices where m = 6 and n = 4*

obtained after a predefined number of iterations. We adopt the later criterion in our experiments.

These steps of a typical GA are shown in Algorithm 1.

Algorithm 1 *The steps in a typical GA*

1: initial population generation
2: evaluation
3: **while** stopping criteria not met **do**
4: selection
5: crossover
6: mutation
7: evaluation
8: **end while**

4.2.2 Solution

Our approaches for the initial population generation, crossover and mutation operators are inspired by the knowledge-augmented genetic algorithm presented in [CFR99] where certain knowledge of the problem is taken into account to generate the initial population and perform crossover and mutation operations.

Encoding of A Solution

We encode a chromosome (ch_i) as a sequence of numbers. The length of the chromosome is the same as the total number of processing tasks, i.e., m. The first number of the chromosome represents the index of the device on which the first task will execute; the second number represents the index of the device on which the second task will execute and so on. The jth number in a chromosome ch_i is denoted as $ch_i(j)$. Thus, every chromosome represents a different assignment. An example of the chromosome corresponding to the assignment illustrated in Figure 4.1 is given in Figure 4.5.

Initial Population Generation

The search space consists of all the task to device mappings following the reachability constraint and closed by the chromosome assigning each task to the first device and the chromosome assigning each task to the last (nth) device. In order to satisfy the reachability constraint, we introduced certain ordering in the initial population using three approaches described as the following to possibly include the members covering a large area of the search space.

1. In the first approach, we assign the first task to a random device with an index of $j < n$. The subsequent tasks are assigned the device whose index is equal to or greater than the device's index associated with the previous task. However, this approach leads to the generation of the chromosomes where most of the tasks are assigned to device n.

2. The second approach chooses a task $b < m$ randomly and assigns a random device $r < n$ to it. The tasks indexed by $i < b$ $(i > b)$ are assigned to the device whose index is equal to or less (greater) than the index of the device assigned to the next (previous) task. This approach generates initial populations with more variety than the first approach. However, this approach leads to the generation of the chromosomes where most of the tasks are assigned device indexed by $1, n$ or a combination of them in a chromosome.

3. The third approach of the initial population generation is more controlled than the second approach. The initial step of choosing a random task and assigning a random device to it remains the same. However, the tasks indexed by $i < b$ $(i > b)$ are assigned to the device whose index is equal to or *just one* less (greater) than the index of the device assigned to the next (previous) task. This approach generates the members of the initial population spreading most of the search space.

Objective Function and Assigning Fitness

Calculating the end-to-end delay (Ω^{de}) of a task assignment encoded by the chromosome (ch_i) is the function as described earlier (Equation 3.3). The fitness f_i of a chromosome ch_i is calculated as follows:

$$\begin{aligned} f_i &= \max(\Omega^{\mathrm{de}}(ch_1), \Omega^{\mathrm{de}}(ch_2), \ldots, \Omega^{\mathrm{de}}(ch_{pop.size})) \\ &\quad -\Omega^{\mathrm{de}}(ch_i) + 1 \end{aligned} \tag{4.3}$$

This ensures that the fitness of a chromosome is inversely related to its end-to-end delay and positive.

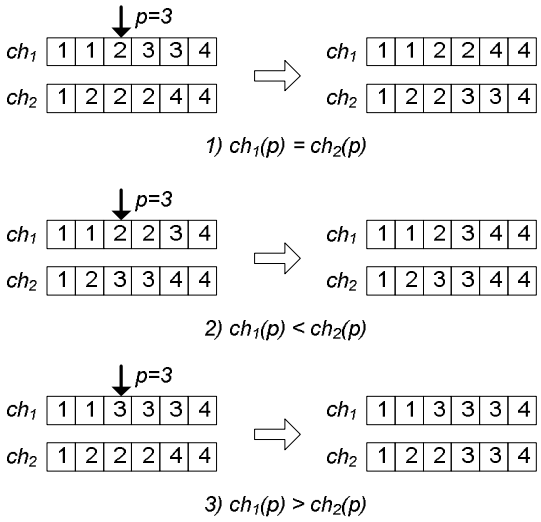

Figure 4.6: *Effect of crossover operator*

Crossover Operator

Let ch_1 and ch_2 be the two chromosomes selected for a crossover. To ensure that the crossover operator results in chromosomes satisfying the reachability constraints, we consider the following three conditions in the given order at the randomly chosen crossover point p:

1. $ch_1(p) = ch_2(p)$: This case results in the normal crossover where ch_1 and ch_2's parts after the crossover point are exchanged.
2. $ch_1(p) < ch_2(p)$: In this case, ch_1's part after the crossover point is replaced with the ch_2's part after the crossover point. There is no change in ch_2.
3. $ch_1(p) > ch_2(p)$: In this case, ch_2's part after the crossover point is replaced with the ch_1's part after the crossover point. There is no change in ch_1.

Figure 4.6 shows the results of the crossover for the three conditions explained above. The probability of crossover in our experiments is 0.8.

Mutation Operator

In a chromosome ch_i, let u be the index of the element chosen for the mutation. To ensure the reachability constraint, the mutation operator changes $ch_i(u)$ to a value which is in the range of [$ch_i(u-1)$,..., $ch_i(u+1)$] inclusive of both the values.

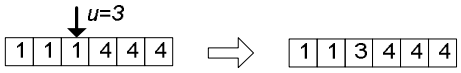

Figure 4.7: *Effect of mutation operator*

Figure 4.7 shows the result of the mutation. The value of 1 could have mutated in the range $[1, ..., 4]$. The probability of mutation in our experiments is 0.01.

4.2.3 Performance Analysis

We conducted a set of experiments to understand better the performance of GA algorithms. In particular, we executed the GA algorithms along with the exact algorithm proposed in Section 4.1 for a combination of various number of tasks and devices. We executed three variants of GA algorithm based on the three initial population generation strategies described in the Section 4.2.2. We refer to these algorithms as GA-1, GA-2 and GA-3. For each GA algorithm, we experimented with two population sizes of $0.5mn$ and mn respectively. The results reported for GA algorithm are averaged over 10 runs each. For example, Table 4.2 shows the comparison results obtained for the GA-3 algorithm with the population size $0.5mn$. Other results are excluded from the table for brevity.

CPU Time

In Figure 4.8 and Figure 4.9, we compare the required execution time of GA-1, GA-2 and GA-3 algorithms for the population size $0.5mn$ and mn respectively. Both results are also compared against the the exact algorithm proposed in Section 4.1. As it can be observed from the Figure 4.8, for the population size $0.5mn$, as the number mn increases, the GA algorithms require less time than the exact algorithm. However, as Figure 4.9 shows, for the population size mn, GA-3 requires higher execution time than the exact algorithm.

GA Solution Quality

As the exact algorithm provides an optimal solution to the assignment problem and GA provides sub-optimal solution, Figure 4.10 and Figure 4.11 compare the average difference in the end-to-end processing delay (a sub-optimal solution) obtained using GA-1, GA-2 and GA-3 algorithms to the optimal solution for different GA population size. These two graphs show that GA-3 algorithm outperforms GA-1 and GA-2 algorithms by a substantial margin. The average percentage difference

Setting			**CPU time** (milliseconds)		**End-to-end delay**		
m	n	pop.size	Exact	Avg. GA-3	Exact	Avg. GA-3	Best GA-3
20	10	100	47	13	1167	1346	1241
30	10	150	32	36	1743	1885	1773
40	10	200	31	39	2195	2572	2425
30	20	300	110	94	2029	2161	2090
40	20	400	156	230	2457	2750	2627
50	20	500	157	303.1	3157	3349.1	3255
40	30	600	391	335.9	2845	3244.9	3100
50	30	750	500	528.1	3357	3720.1	3650
60	30	900	875	665.6	3732	4379.7	4105
60	40	1200	1344	1310.9	3825	4351.6	4072
70	40	1400	2015	1779.8	4418	4939.4	4800
70	50	1750	3109	1079.7	4792	6069.6	5555
80	50	2000	4125	3689.1	5257	6081.9	5719
80	60	2400	7125	5201.4	5463	6573.5	6454
90	60	2700	7937	5782.8	6147	7148.3	6923
90	70	3150	10985	8988.8	6111	6871.7	6643
100	70	3500	13296	13034.1	6400	7271.4	6991
100	80	4000	23437	13245.2	7261	8331.7	8098

Table 4.2: *Some of the results comparing the performance of GA with the exact approach*

in the sub-optimal solution obtained using GA-3 algorithm is around 10% to the optimal solution.

4.3 Tree-Star Assignment

As motivated by Bokhari [Bok88], many distributed application settings can be modeled as a mapping of a task tree onto a star network (c.f. Section 2.2.3). In MPMSs, we expect this tree-star model to be popular as well. However, it is not possible to apply those existing algorithms onto our problem directly due to many assumptions and constraints associated with it. In this section, we present a new tree-star assignment algorithm and our work differs from Bokhari's original approach and those follow-up studies in the following two aspects:

1. The following three constraints are relaxed in our algorithm by a coloring

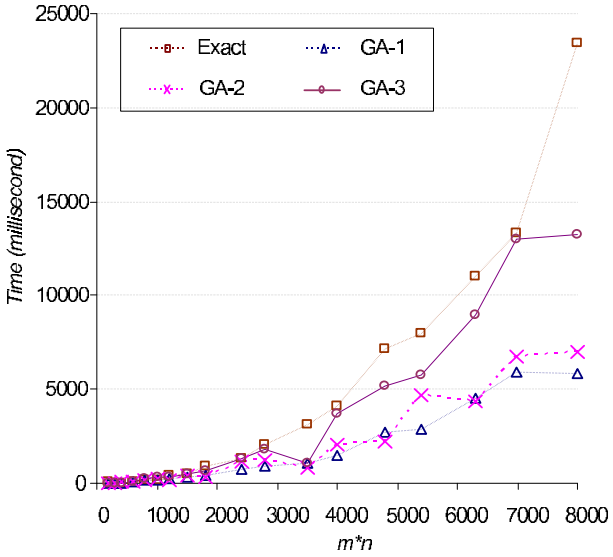

Figure 4.8: *Time required to obtain the (sub) optimal solutions. For the GA algorithms, the population size is* $0.5mn$

scheme:

- There are as many satellite devices as there are leaf vertices in the task tree and it is possible not to use them if the optimal assignment dictates so, i.e., partitioning on the task tree is done first and then the leaf tasks are located on the satellite devices based on the result. This is main reason why the existing algorithm can not be applied to our problem.
- All the satellite devices are homogeneous.
- If two tasks are assigned to a satellite device, their lowest common ancestor is also assigned to the same satellite device.

2. Bokhari proposed the Sum-Bottleneck (SB) algorithm to find a partition that minimizes the bottleneck processing time while our goal is to find a partition that minimizes the end-to-end delay. We propose the Summation of S weight and B weight (SSB) algorithm to tackle this different objective.

4.3.1 Formulation

We formulate the problem of optimal assignment of a task tree onto a star-like device network as:

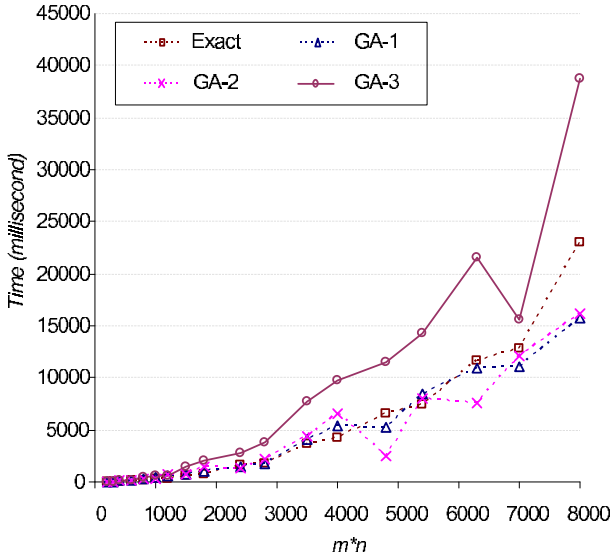

Figure 4.9: *Time required to obtain the (sub) optimal solutions. For the GA algorithms, the population size is* mn

- **Given**: (1) an m-health platform modeled as a star network consisting of a host device and several satellite devices; (2) a telemonitoring application modeled as a processing task tree where each leaf task is a special sensing task. (3) A performance evaluation function $Omega^{\text{de}}$ for task assignments.
- **Goal**: to find an optimal assignment of tasks to the host and satellite devices such that it results in a minimum end-to-end processing and communication delay, i.e., to minimize the summation of maximum processing delay spent at the satellite devices (including the transmission delay to transmit the processed result from the satellite to the host) and the processing delay required at the host device.
- **Subject to**: The root task is fix associated with the host device and each leaf task is fix associated with one satellite device.

4.3.2 Doubly Weighted Graph (DWG)

In this section, we study a DWG and introduce a measure of paths in the DWG, the SSB weight. An algorithm is proposed to search for the optimal path that has minimum SSB weight.

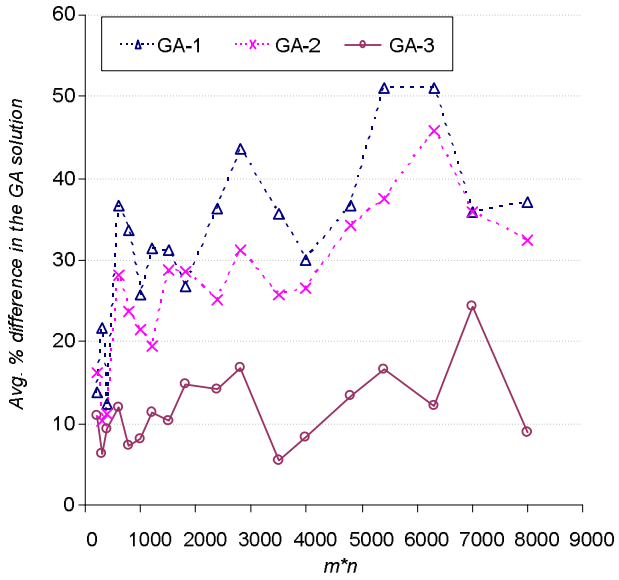

Figure 4.10: *The average difference in percentage for the sub-optimal solution obtained using GA as compared to the optimal solution. For the GA algorithm, the population size is* $0.5mn$

DWG and SSB weight

A DWG (Doubly Weighted Graph) $G = (V, E)$ has two ordered non-negative weights associated with each edge e of E, for example: a sum weight $\sigma(e)$ and a bottleneck weight $\beta(e)$. We define further an S weight and a B weight of a path p that connects two distinguished vertices in G as $S(p)$ and $B(p)$ respectively, which are defined as:

$$S(p) = \sum_{e_i \in p} [\sigma(e_i)] \tag{4.4}$$

$$B(p) = \max_{e_i \in p} [\beta(e_i)] \tag{4.5}$$

Now we introduce the measure of the SSB weight of a path p as $SSB(P)$ where $SSB(p) = S(p) + B(p)$. The optimal SSB path in a DWG is defined as the path with a minimum SSB weight. This SSB weight is therefore a different measure compared to the SB weight studied by Bokhari, where the SB weight of a path p is defined as $\max(S(p), B(p))$.

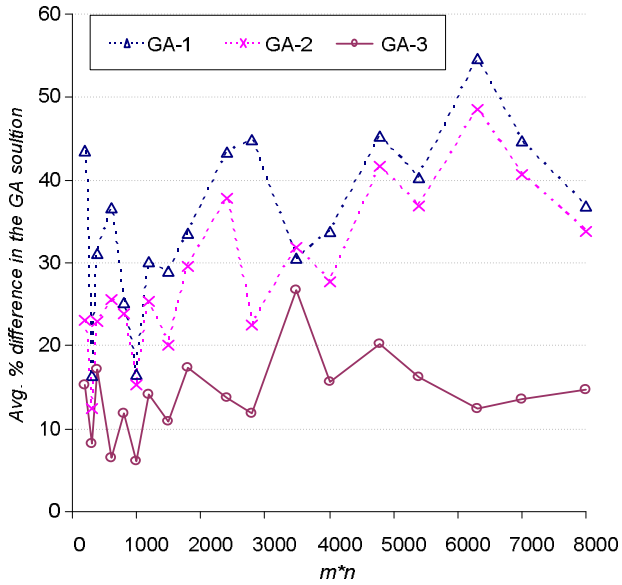

Figure 4.11: *The average difference in percentage for the sub-optimal solution obtained using GA as compared to the optimal solution. For the GA algorithm, the population size is* mn

Algorithm for finding the optimal SSB path

Inspired by the earlier discussions on DWG [Chr75], we present the SSB algorithm for finding the optimal SSB path connecting two distinguished vertices, i.e., "S" and "T", in a DWG. This algorithm works by recording the candidate optimal SSB paths and progressively eliminating edges from the graph when they cannot be a part of the optimal SSB path, until the graph becomes disconnected or further searching will definitely not yield any better path. The pseudo code of this algorithm is presented in Algorithm 2.

We briefly explain how the SSB algorithm works in the following:

1. Prior to the start, two state variables are initiated: the candidate optimal SSB path is set as null and its SSB weight is set as $+\infty$.

2. In each iteration, the algorithm first searches for a path p in G with the minimum S weight. A shortest path searching algorithm can be applied for this purpose. If the SSB weight of p is smaller than the SSB weight of the currently recorded optimal SSB path, the recorded optimal SSB path and its weight will be updated with the new path p. At the end of each iteration, G is updated by removing all the edges whose bottleneck weight is equal to or greater than

Algorithm 2 *SSB algrithm*

```
1: G is a DWG with two distinguished vertices: "S" and "T"
2: optimal_SSB_path ⇐ null
3: optimal_SSB_weight ⇐ +∞
4: while G is connected ∧optimal_SSB_weight > S(p) do
5:    find a path p, the shortest S weight path in G connecting "S" to "T"
6:    if SSB(p) < optimal_SSB_weight then
7:       optimal_SSB_weight ⇐ SSB(p)
8:       optimal_SSB_path ⇐ p
9:    end if
10:   update G by removing all the edges whose bottleneck weight ≥ B(p)
11: end while
12: return optimal_SSB_path
```

$B(p)$. The reason that we can safely remove these edges is that because all of them except for those that belong to the currently recorded optimal SSB path are for sure not part of the optimal SSB path in the DWG. Therefore, at the end of this iteration, either the reduced G contains the optimal SSB path of the original DWG or the current recorded optimal SSB path is the real optimal one.

3. The iterations continue until either the new G becomes disconnected or the S weight of p is greater than the SSB weight of the currently recorded optimal SSB path, i.e., all the remaining paths' SSB weights are greater than the one of the currently recorded optimal SSB path. Thus, the currently recorded optimal SSB path is the optimal SSB path in the original DWG.

Let $|V|$ denote the number of vertices and $|E|$ denote the number of edges in the considered in the DWG. Each iteration in the SSB algorithm applies a shortest path searching with the complexity of $O(|V|^2)$ if the Dijkstra algorithm is used. In the worst case, $|E|$ times of iteration are required, i.e., eliminating one edge per iteration. Thus, the total time complexity of this algorithm is $O(|V|^2|E|)$. An example of finding an optimal SSB path by applying the proposed algorithm is illustrated in Figure 4.12. Given this simple DWG, three iterations are executed to identify an optimal SSB path ("$< 5, 10 > - < 5, 10 >$") with the SSB weight of 20.

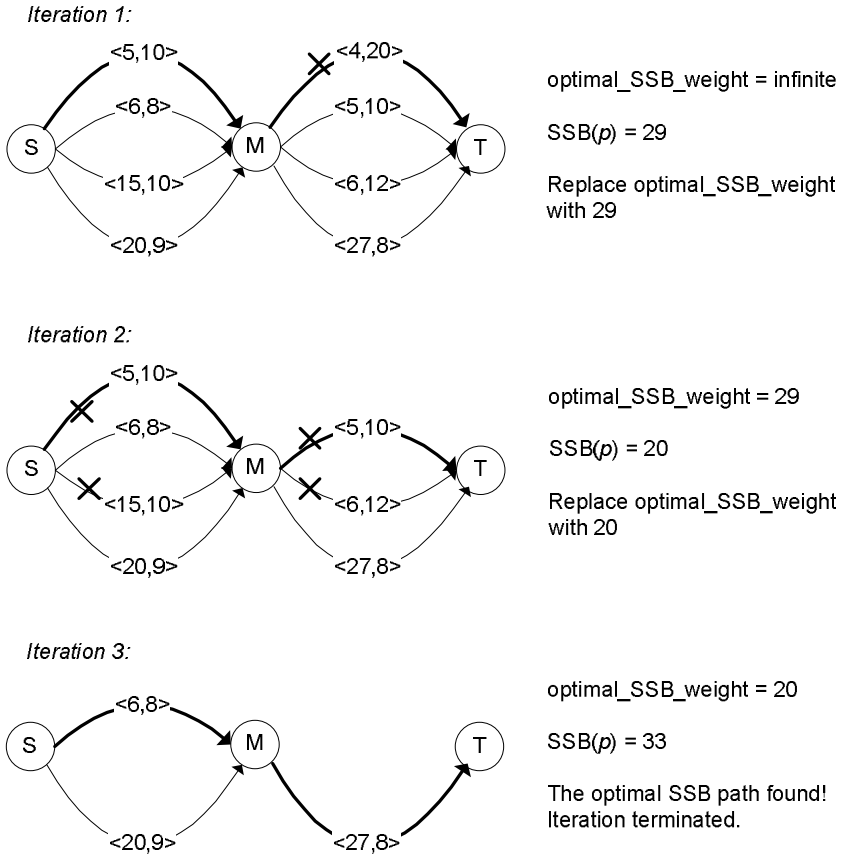

Figure 4.12: *An example of searching for the optimal SSB path, where the thick path indicates the newly found S weight shortest path and the cross indicates the paths to be eliminated*

4.3.3 Solution

In this section, we present the step by step solution of finding the optimal assignment in a tree-star assignment problem as illustrated in Figure 4.13.

Coloring and Labeling The Task Tree

First, we paint each satellite device in the given device star network with a distinguishable color, e.g., "red" for satellite "R", "yellow" for satellite "Y", "blue" for satellite "B" and "green" for satellite "G". Then, in the given task tree, every leaf sensing task inherits the color from its associated satellite device and each edge is painted by "propagating" the color from the leaf tasks towards the root task (Fig-

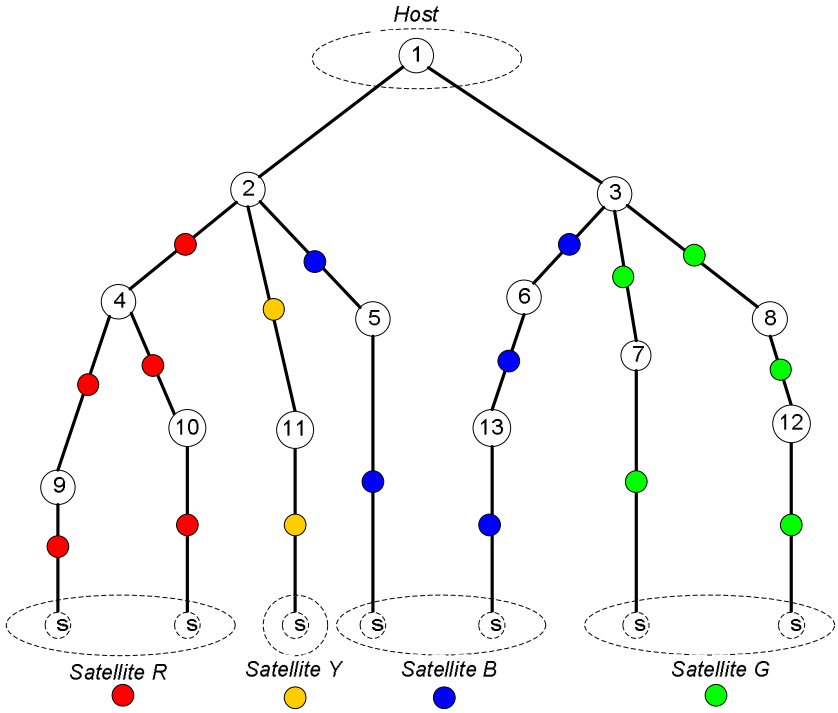

Figure 4.13: *Coloring the edges in a task tree*

ure 4.13). The exceptions to this coloring are the edges of "$p_1 - p_2$" and "$p_1 - p_3$" since the propagated colors conflict. This phenomenon implies that p_1, p_2 and p_3 have to be deployed on the host device, since they need to process the information obtained from multiple satellites.

The next step is to doubly label the task tree with a sum weight (σ) and a bottleneck weight (β) on each edge. The detailed procedure is the same as the original routine proposed by Bokhari (c.f. Section 2.2.3).

Building The Colored Assignment Graph

Similar to Bokhari's approach, in the task tree, all the sensing tasks are merged into a single dummy task vertex "$\triangle$". The vertices which will constitute to the assignment graph (squares in Figure 4.14) are inserted in each face of the task tree and on the left and right-hand sides of the tree. An directed assignment graph of this modified tree is now drawn by adding an edge between every pair of vertices that belong to faces that have a common colored tree edge. The edge of this assignment graph inherits

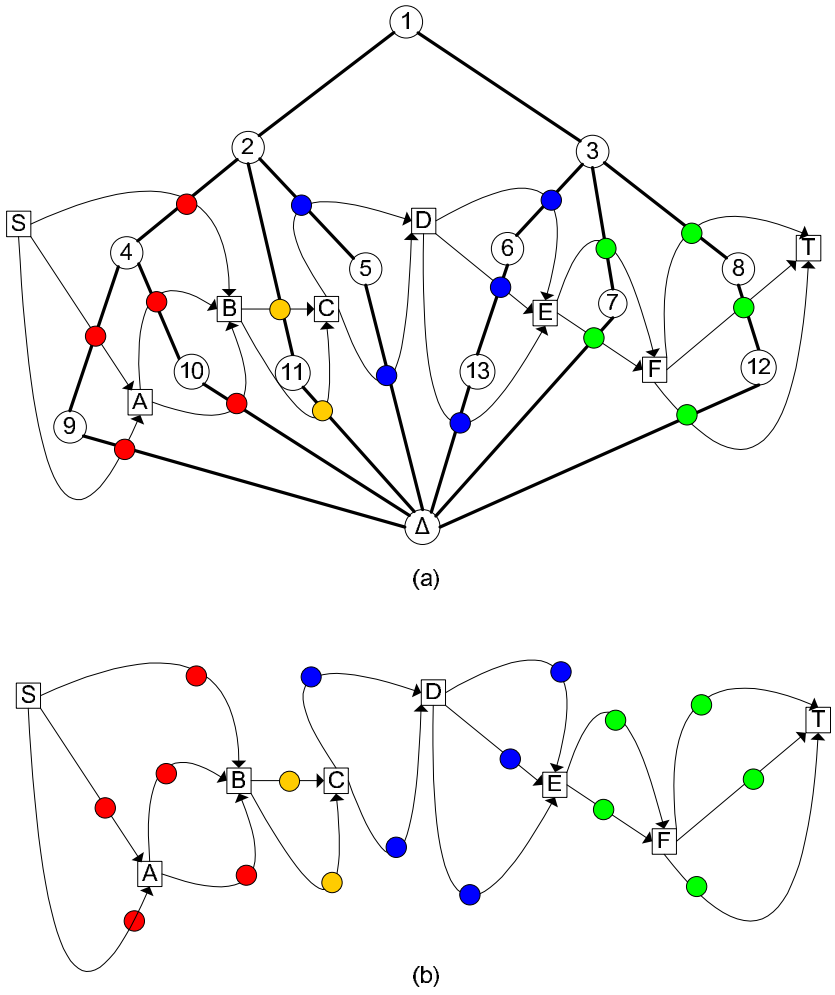

Figure 4.14: *(a) illustrates the procedure of building the colored assignment graph; (b) presents the assignment graph by eliminating the original tree*

the color and the double weights of the tree edge it crosses. This procedure and the resulting colored assignment graph are shown in Figure 4.14.

Finding The Optimal SSB Path

Now, we have built a colored doubly weighted assignment graph (Figure 4.14 (b)). Each path connecting the vertices "S" and "T" in this graph corresponds to an assignment of the task tree on the star network. The end-to-end delay of this partition equals to the colored path's SSB weight, i.e., the summation of the colored path's S weight and B weight. The colored path's S weight is defined in the same way as the non-colored DWG, i.e., $S(p) = \sum_{e_i \in p}[\sigma(e_i)]$. The colored path's B weight is defined as the maximum among the summations of the bottleneck weights per color:

$$B(p) = \max_{e_i \in p}[\sigma_{\text{red}}\beta(e_i), \sigma_{\text{yellow}}\beta(e_i), \sigma_{\text{blue}}\beta(e_i), \ldots] \tag{4.6}$$

In order to identify the optimal assignment, it is required to search for the colored path with minimal SSB weight. The previously proposed SSB algorithm can be adapted to serve for this purpose. When the B weight of the shortest-path (determined by paths' S weight) is contributed by the subsequent edges having the same color, that part of the assignment graph should be expanded before any edges are eliminated. This ensures that during the edge removal step, only the edges which do not contribute to the optimal SSB path anymore are removed. For example, as shown in Figure 4.15, if the B weight of the shortest path (the topmost path) is a sum of the bottleneck weights of the two blue edges (labeled "b1" and "b2"), the entire blue part of the graph should be expanded into a number of edges, each of which represents a possible path between vertex "C" and "E". Therefore, in this specific case, due to the graph expansion, the running time of the adapted SSB algorithm is in the order of $O(|V|^2|E'|)$, where $|E'|$ is the number of edges in the expanded graph.

4.4 Concluding Remarks

This chapter proposes and evaluates task assignment algorithms for situation with specific topological models of the underlying system and the processing task graph. First, we present an efficient task assignment algorithm when both the telemonitoring application and the m-health platform form a chain and the goal of the task assignment is to find an optimal assignment with minimal end-to-end delay. Compared to earlier work, this new method relaxes a so-called contiguity constraint, which is a necessity in the earlier case but unnecessarily constraining MPMSs. Furthermore, we observed from our experiments that the assignment graph "construction" step is more time consuming than the actual "search" step in most of cases,

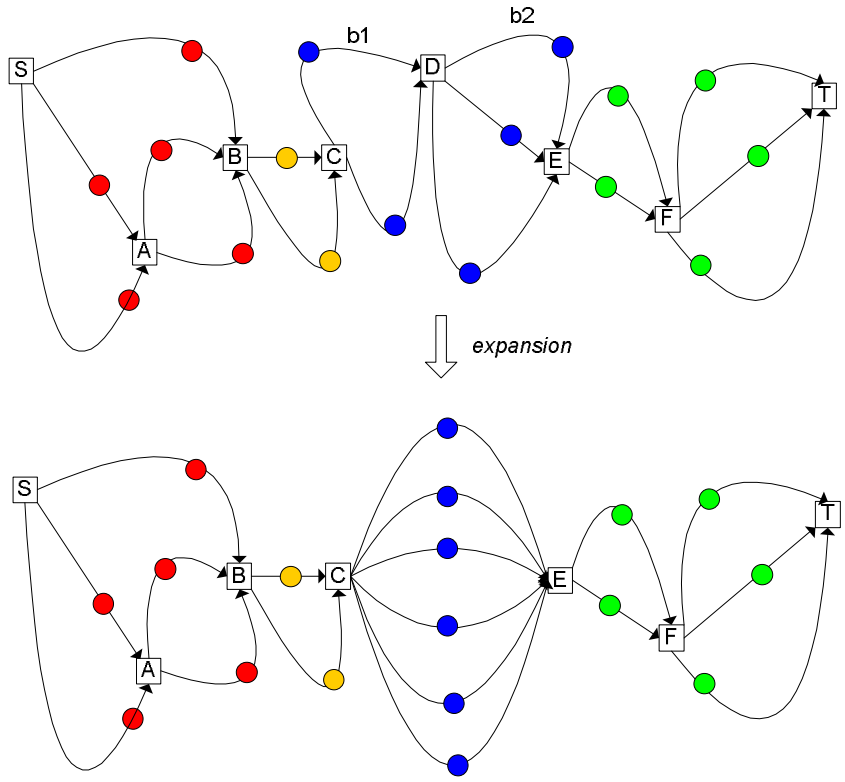

Figure 4.15: *Expanding part of the colored assignment graph*

which suggests that the evaluation on this kind of graph-based method should be based on the time complexity of the entire method instead of the "search" step only [Bok88, SC90, NO91]. Although the graph-based exact algorithm is more efficient compared to earlier work, it still has a major drawback on memory consumption. Hence, we also propose a GA based solution that has much smaller and bounded requirements on memory space while providing satisfactory results. At last, we study another variant of the task assignment problem in tree-star model. Again, our method relaxes several constraints existed in earlier work and tackles the new objective on minimizing end-to-end delay. However, due to the complication and limit time, we did not implement the proposed tree-star algorithm and thus no performance experiment is reported. As a workaround, people who are interested in the tree-star problem can use A*-based algorithms introduced in Chapter 5.

Chapter 5

A*-Based Task Assignment Algorithms

In this chapter, we focus on the task assignment problem in its general form (DAG-DAG) as formulated earlier (c.f. Section 3.3.3). In many MPMSs, only a limited number of tasks and resources are involved. For example, the epilepsy detection scenario presented in Section 1.1.2 deals with 10 processing tasks and less than 10 devices. Thus, it is computationally feasible to find the optimal task assignment by exact algorithms. The well-known A*-algorithm is used as the foundation of our solution. This decision is motivated by the fact that the A*-algorithm guarantees a better performance, e.g., generating a smaller search tree, than any other search algorithms with the same admissible heuristic [DP85].

This chapter is structured as follows. Section 5.1 explains the basic concepts of A*-algorithm and the general design of A*-based task assignment algorithms. Based on the general problem formulation, we categorize three problem types that are different in their objective functions: maximizing system battery lifetime, minimizing end-to-end delay, and maximizing application availability. In Section 5.2, we present the specific designs with regard to the problem of maximizing system battery lifetime and the performance evaluation of the proposed algorithm. Section 5.3 concentrates on the problem of minimizing end-to-end delay. The third type was studied extensively earlier by Franken [Fra96], thus we omit it in this thesis and refer interested readers to the earlier work.

5.1 Design of A*-Based Task Assignment Algorithms

This section explains the basic concepts of the A*-algorithm and the general design of A*-based task assignment algorithms.

5.1.1 The A*-Algorithm

The A*-algorithm is a "best-first"[1] graph search algorithm that finds the least-cost path from a given initial node to one goal node (out of one or more possible goals) [RN02]. Through constructing and traversing a search tree that represents the solution space of a particular problem, the A*-algorithm finds the optimal solution with the lowest cost. In this search tree, the root node represents a null solution, an intermediate node represents a partial solution and a leaf (goal) node represents a complete solution. A high-level graphical view of the A* search tree is depicted in Figure 5.1. Each node, n, has an associated cost function $f(n)$ that is a sum of two functions $g(n)$ and $h(n)$: $g(n)$ is the actual cost of this partial solution and $h(n)$ is a lower-bound estimation of the additional cost from n to the leaf node. The A*-algorithm maintains a sorted list ($OPEN$) containing all nodes that are either discovered leaf nodes or intermediate nodes that can still be expanded. The algorithm proceeds by always removing the node in the $OPEN$ list with a minimal $f(n)$ and expanding the search tree from this node. The iterations terminate when the retrieved node from the $OPEN$ list is a leaf node and this leaf node represents the optimal solution. The A*-algorithm a very efficient search approach since it can avoid traversing the entire search tree, i.e., some branches are not constructed at all. This is because we know that, based on the calculation of $f(n)$, none of those branches will lead to an optimal solution. The pseudo-code of the A*-search algorithm is illustrated in Algorithm 3.

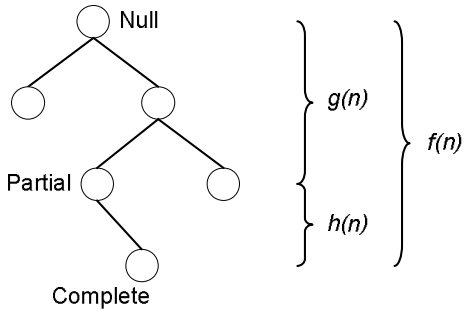

Figure 5.1: *The illustration of an A* search tree.*

The A*-algorithm defines a best-first search approach without detailing the search tree construction and the functions $g(n)$ and $h(n)$. Thus, if we want to apply the A*-algorithm to solve a specific problem, these two parts need to be addressed based

[1] A "best-first" search algorithm is an algorithm that traverses a graph by expanding the most promising node determined based on a specified rule.

Algorithm 3 *High level pseudo-code of A* algorithm*

```
1: initialize a sorted list OPEN
2: add a root node with f(root) := 0 into OPEN
3: while true do
4:    remove node n with lowest f(n) from OPEN
5:    if n is a goal node then
6:       return n as the optimal solution
7:    end if
8:    expand from n to a set of child nodes {cn_i}
9:    for all nodes in cn_i do
10:      calculate g(cn_i), h(cn_i)
11:      f(cn_i) := g(cn_i) + h(cn_i)
12:      add cn_i with f(cn_i) into OPEN
13:   end for
14: end while
```

on the specific problem. In the next section, we will first explain how to construct the search tree for the task assignment problem in MPMSs. The formulation of functions $g(n)$ and $h(n)$ will be discussed later when the two problem types are studied (c.f. Section 5.2.2 and Section 5.3.2).

5.1.2 Search Tree Construction

This section presents the search tree design for solving the task assignment problem in MPMSs. First, we explain what each node in the search tree represents and how to encode task assignments into the search tree. Second, we introduce the concept of task search order and its impact on the algorithm performance. Third, we propose two sets of specific designs, namely node expansion rules and data dependence check, to ensure the traversed search tree to be both effective and efficient. By "effective", we mean the search tree should not miss a task assignment that offers satisfactory performance. By "efficient", we mean the search should avoid the unnecessary parts of the search tree as much as possible.

Node Representation in Search Tree

To solve the task assignment problem as formulated earlier (c.f. Section 3.3.3), we first need to consider how to construct a search tree that represents all candidate task assignments. In general, we follow an approach similar to some earlier work on A*-based task assignment algorithms [RCD91, Fra96, KA98, HM05]. In our search tree,

the root node represents an empty-assignment, every intermediate node represents a partial-assignment and every leaf node represents a full-assignment. When an intermediate node (or root node) is expanded to its child nodes, a new task is assigned to a proper resource. For example, in the search tree presented in Figure 5.2, each node is labeled with the new assignment of a task to a resource: (1) the node of "$p_1 \rightarrow d_1$" represents the partial-assignment of the processing task "p_1" to the device "d_1"; (2) the node of "$t_1 \rightarrow d_1, c_3, d_3$" represents the partial-assignment of the transmission task "t_1" to the communication path "d_1, c_3, d_3".

Task Search Order

Task search order is an important concept in task assignment problem and determining the task search order is an essential step in constructing a search tree. As discussed in earlier research [RCD91], different orders potentially have large impact on the search tree size, hence, on the algorithm performance. If we can order tasks such that, at shallow tree levels, less number of nodes are expanded and the cost difference between the expanded nodes are larger, then a smaller search tree will be generated by the A*-algorithm. We examine the possibility by designing several strategies to order the tasks and testing their performance in the experiment sections. To support constructing a search tree such that the reachability constraint can be efficiently guaranteed, we define two general rules to order tasks based on the directions in a task DAG:

1. a task can only be assigned, i.e., to expand the search tree by assigning it to all possible devices or communication paths, when at least one of its direct predecessor tasks has been assigned already.

2. if a transmission task is assigned, then its direct successor (processing) task must be assigned immediately if it is not assigned yet.

Given these general rules, a number of search orders are possible for a task DAG. For example, a depth-first task search order for the problem illustrated in Figure 3.4 (or its reproduction Figure 5.3) is: $\{p_1, t_1, p_2, t_2, p_3, t_3, p_8, t_4, p_4, t_5, p_5, t_6, p_6, t_7, p_7, t_8\}$. An example of breath-first task search order for the same problem is: $\{p_1, p_5, t_1, p_2, t_6, p_6, t_2, p_3, t_4, p_4, t_7, p_7, t_3, p_8, t_5, t_8\}$. The search tree generated from the depth-first task search order is presented in Figure 5.2.

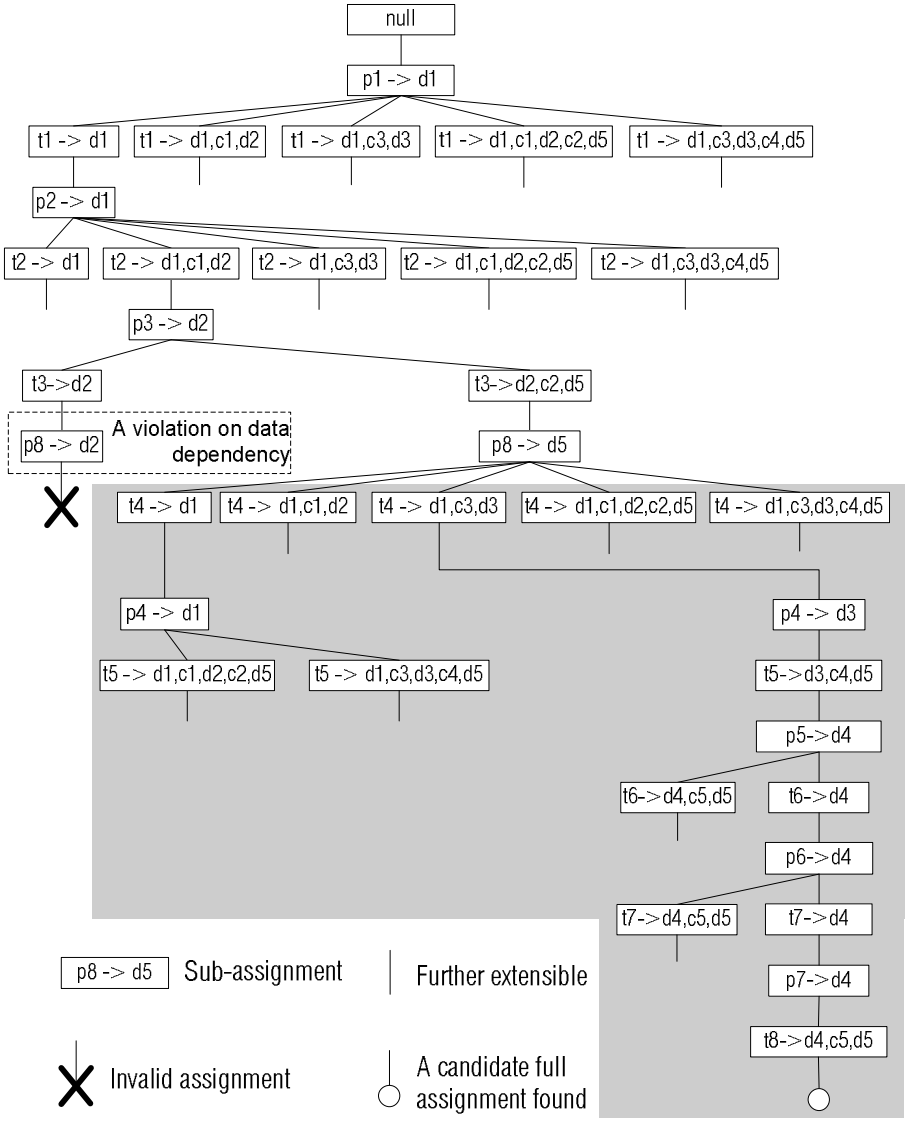

Figure 5.2: *Search tree (partial view) for the problem shown in Figure 3.4 (or Figure 5.3) with a depth-first task search order. For clarity, each node is only labeled with the assignment of latest assigned task.*

Node Expansion Rules

Following a defined task search order, when a new task is assigned, the corresponding node in the search tree should be expanded to represent the new partial-assignments. Due to the reachability constraint, type constraint and local constraint, there are a limited number of resources to which this new task can be assigned. To ensure the node expansion is performed properly, we propose a set of node expansion rules in the following. In our explanation, we refer to a newly to-be-assigned task as i and use some nodes presented in Figure 5.2 as examples.

1. When i is a processing task, there are two possibilities:
   - If i is a source processing task, then it is assigned to all possible devices subject to its local constraint, e.g., "$p_1 \rightarrow d_1$".
   - If i is *not* a source processing task, then one of the direct predecessor (transmission) tasks of i must be assigned already to a communication path, denoted as cp_α. This is because of the first general rule we enforced on ordering tasks. In this case, the search tree must be expanded by assigning i to the ending device of cp_α, e.g., "$p_3 \rightarrow d_2$".
2. When i is a transmission task, we also distinguish between two cases. The direct predecessor (processing) task of i must have been assigned to a device already due to the first "general rule" for ordering tasks and we denote this device as d_α.
   - When i's direct successor (processing) task has not been assigned yet, there will be some freedom to assign i: we find all directed communication paths starting with d_α and assign i to each of these paths, e.g., "$t_3 \rightarrow d_2$" and "$t_3 \rightarrow d_2, c_2, d_5$". Note that we treat a single device as a special communication path (c.f. Section 3.3.1).
   - If i's direct successor (processing) task has already been assigned to a device vertex (d_β), i has to be assigned to a directed communication path connecting device d_α with device d_β, e.g., "$t_8 \rightarrow d_4, c_5, d_5$". If it is not possible to find such a path, then the current partial-assignment is not valid and this branch of the search tree has to be pruned.

Data Dependency Check

Due to the reachability constraint modeled in this thesis, a number of assignments will become invalid eventually. In order to perform an effective and yet efficient search for candidate assignments, it is ideal to only generate the branches that lead

to at least one valid full-assignment. However, this requires a complicated and impractical design since the algorithm should, upon assigning every new task, "foresee" the unassigned tasks and their assignment constraints. Instead of striving for such a "perfect" search tree, we propose a method of data dependency check to avoid traversing some of the branches that will not lead to any valid full-assignments due to the violation of reachability constraint.

In order to explain the data dependency check, we take the problem presented earlier in Figure 3.4 as an example. In this problem, due to the fixed association of sensory tasks with sensors, "p_1" has to be assigned to "d_1", and "p_5" has to be assigned to "d_5". The data dependency check is performed as follows. First, we examine the sensory data reachability in the task DAG and label each task with its dependency on sensory data (obtained by a source processing task) as shown in Figure 5.3. For example, "t_1" and "p_2" are labeled with "p_1" to indicate that they are dependent on the input data from the source processing task "p_1". Second, in the resource DAG, each resource can be labeled with its reachability of a particular sensory data. Then, by comparing the data dependency in a task DAG and the data reachability in a resource DAG, we can determine whether a particular partial-assignment is valid or not. For example, from Figure 5.3, it is clear that "p_8" can only be assigned to "d_5": since "p_8" requires sensory data from "p_1" and "p_5" and only "d_5" satisfies this data reachability requirement.

The benefit of this data dependency check could be very substantial. For example, as illustrated in Figure 5.2, a violation can be identified at the node "$p_8 \rightarrow d_2$" if we apply the data dependency check and this branch can be safely pruned here. Otherwise, the node "$p_8 \rightarrow d_2$" has to be extended further and an entire branch identical to the one highlighted in the grey area has to be constructed. Only at the final step when "t_8" is assigned, all the branches extended from the node "$p_8 \rightarrow d_2$" can be identified as invalid since no directed communication path can be found connecting from the hosting device of "p_7" to "d_2".

Again, we must admit that although this data dependency check can identify some invalid assignments in an earlier stage, it still can not produce a "perfect" search tree. For example, in Figure 5.2, when the node "$t_3 \rightarrow d_2$" is traversed, we could already identify this branch as non-valid if we could "foresee" the assignment constraint of unassigned task, i.e., "p_8". Nevertheless, by enabling the data dependency check, A*-based task assignment can compute candidate assignments more efficiently compared to the version without this check.

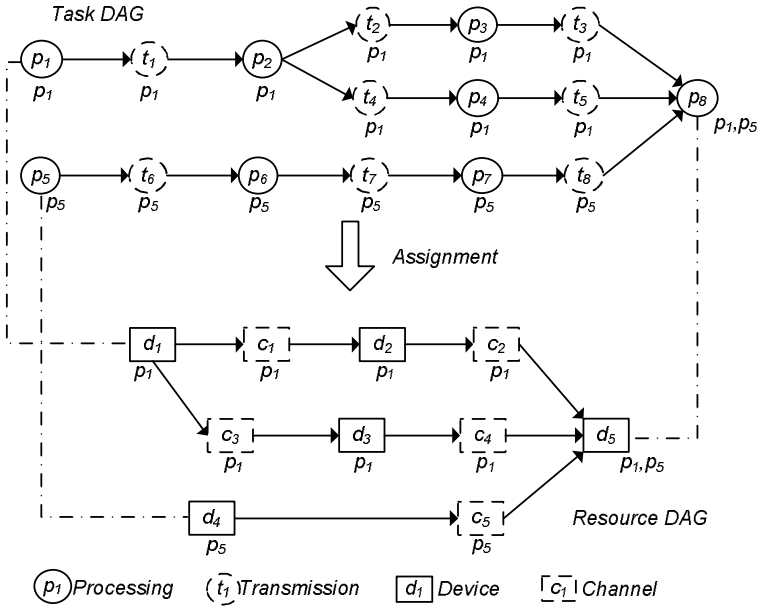

Figure 5.3: *The labeling of data dependency for the problem shown in Figure 3.4. Each task is labeled with its dependence over a particular sensory task. For example, "t_3" depends on "p_1" and "p_8" depends on "p_1, p_5". Each resource is labeled with its reachability of a particular sensory task. For example, "c_2" can be reached by "p_1" and "d_5" can be reached by "p_1, p_5"*

5.2 Maximize System Battery Lifetime

This section proposes an A*-based task assignment algorithm with the objective of maximizing the system battery lifetime. We also evaluate this algorithm's performance based on a Java implementation.

5.2.1 Formulation

Due to the mission-critical nature of many telemonitoring applications, such as in the epilepsy detection application, there is often a strict requirement on end-to-end delay. Furthermore, it is always desirable to increase the system battery lifetime since this means that the patient can have a greater freedom. Thus, in this section, we treat the system battery lifetime as the objective function and the maximum allowed end-to-end delay and the minimum allowed system availability as hard restriction. Very similar to the general formulation of task assignment problem in

MPMS, the task assignment problem on maximizing system battery lifetime is formulated as follows:

- **Given**: (1) a telemonitoring application (P, T, A_t, L_P, L_T); (2) an m-health platform (D, C, A_r, L_D, L_C); (3) Three performance evaluation function Ω^{li}, Ω^{de} and Ω^{av} for task assignments that represent battery lifetime, end-to-end delay, and availability.
- **Goal**: To find a number of candidate assignments $\{\Phi_m^{\text{can}} | m = 1, 2, N_{rc}\}$ among all possible task assignments for which the values of $\Omega^{\text{li}}(\Phi_m^{\text{can}})$ are maximized.
- **Subject to**: four assignment constraints namely *type constraint*, *local constraint*, *assurance constraint*, and *reachability constraint*.

The *assurance constraint* defined here is slightly different with the previous general definition (c.f. Section 3.3.3) by removing the minimum required system battery lifetime. The other three assignment constraints are the same as in the general definition. The high-level routine of the A*-based task assignment algorithm with the objective of maximizing the system battery lifetime is illustrated in Algorithm 4.

5.2.2 Function $f(n)$

The system battery lifetime is the minimum among all devices' battery lifetime, i.e., the battery lifetime of the "bottleneck" device. The value of this cost function decreases monotonically or remains the same when the search tree is expanding. This is because more energy will be consumed by the newly assigned task. Based on the definitions in Section 3.3.1, we explain how to estimate the cost function $f(n)$ as the upper bound system lifetime with the consideration of unassigned tasks. Here, we do not follow the usual practice of A* search approach to define function $g(n)$ and $h(n)$ separately. The reason lies in using system battery lifetime as the cost function, which is not an additive function: If we model $g(n)$ and $h(n)$ to represent the system battery lifetime contributed by the assigned tasks and unassigned tasks separately, their simple summation does not represent the estimation of total system battery lifetime.

We define, for every device d_i in a partial-assignment, a set K_{d_i} containing all the assigned processing and transmission tasks at d_i, a set R_{d_i} containing all the unassigned transmission tasks that are *pointing* to a processing task in K_{d_i} and a set S_{d_i} containing all the unassigned transmission tasks that are *leaving* from a processing task in K_{d_i}. The energy consumption rate at d_i resulting from K_{d_i} is denoted as $k(d_i)$. The minimal energy consumption rate at d_i caused by the tasks in set R_{d_i} and S_{d_i} once they are fully assigned is denoted as $u(d_i)$. The calculations of $k(d_i)$ and

Algorithm 4 *Pseudo code of A*-based task assignment algorithm for maximizing the system battery lifetime*

1: input the number of required candidate assignments as N_{rc}
2: input the minimal required availability $min\Omega^{\mathrm{av}}$ and the maximal allowed end-to-end delay $max\Omega^{\mathrm{de}}$
3: determine a search order for the tasks: $order$
4: initialize a sorted list $OPEN$ to contain the already visited partial-assignments
5: initialize a sorted list $candidates$ to contain the already found candidate assignments
6: create a partial-assignment $root$, $f(root) := 0$, $root.toBeAssignedTask := order.first$ /* For a partial-assignment n, $n.toBeAssignedTask$ contains the next to be assigned task according to $order$*/
7: add $root$ into $OPEN$
8: **while** true **do**
9: **if** $OPEN$ is empty **then**
10: **return** $candidates$ /* Found candidate assignments are less than the required number N_{rc}*/
11: **end if**
12: remove a partial-assignment n with highest $f(n)$ from $OPEN$ /* Currently, the most promising partial-assignment */
13: **if** n is a full-assignment **then**
14: add n into $candidates$ /* Found a new candidate */
15: **if** $candidates.size() = N_{rc}$ **then**
16: **return** $candidates$ /* A list of N_{rc} found candidate assignments */
17: **end if**
18: **end if**
19: expand from n to a set of child partial-assignments $\{cn_i\}$ by assigning $n.toBeAssignedTask$
20: $index := order.getIndex(n.toBeAssignedTask)$ /* Record the index of $n.toBeAssignedTask$ */
21: **for all** partial-assignments in $\{cn_i\}$ **do**
22: **if** no data dependency violation at cn_i **then**
23: **if** $\Omega^{\mathrm{de}}(cn_i) \leq max\Omega^{\mathrm{de}} \wedge \Omega^{\mathrm{av}}(cn_i) \geq min\Omega^{\mathrm{av}}$ **then**
24: /* cn_i is a valid partial-assignment, we should record it in $OPEN$ */
25: calculate $f(cn_i)$
26: $cn_i.toBeAssignedTask := order.get(index + 1)$ /* Record the next to be assigned task */
27: add cn_i into $OPEN$
28: **end if**
29: **end if**
30: **end for**
31: **end while**

$u(d_i)$ are shown in Equation 5.1 and Equation 5.2 respectively. For an overview of symbol definitions, we refer the readers to Table 3.1 and Table 3.2.

$$\begin{aligned} k(d_i) \;=\; & e^{\mathrm{HK}}_{d_i} \\ & + e^{\mathrm{op}}_{d_i} \cdot \sum_{p_j \in K_{d_i}} n^{\mathrm{P}}_{p_j} \\ & + \sum_{\{c_j | (c_j, d_i \in A_r)\}} (e^{\mathrm{R}}_{c_j} \cdot \sum_{\{t_k | c_j \in \Phi(t_k), t_k \in K_{d_i}\}} n^{\mathrm{T}}_{t_k}) \\ & + \sum_{\{c_j | (d_i, c_j \in A_r)\}} (e^{\mathrm{S}}_{c_j} \cdot \sum_{\{t_k | c_j \in \Phi(t_k), t_k \in K_{d_i}\}} n^{\mathrm{T}}_{t_k}) \end{aligned} \tag{5.1}$$

The energy consumption rate at d_i resulting from the assigned tasks is the summation of four terms.

- The 1st term is the energy consumption rate of the "housekeeping" activities at device d_i, e.g., display, network interface cards, etc.
- The 2nd term is the energy consumption rate caused by all assigned processing tasks, where $e^{\mathrm{op}}_{d_i}$ is the energy consumption of one operation at device d_i and $n^{\mathrm{P}}_{p_j}$ is the number of operations per time unit of processing task p_i.
- The 3rd term is the energy consumption rate caused by all assigned transmission tasks on d_i's receiving end, where $e^{\mathrm{R}}_{c_j}$ denotes the energy consumption of receiving one data unit through channel c_i, $n^{\mathrm{T}}_{t_k}$ denotes the number of transmitted data units per time unit of transmission task t_i, $\{c_j | (c_j, d_i) \in A_r\}$ is the set of channels that are pointing to device d_i, and $\{t_k | c_j \in \Phi(t_k), t_k \in K_{d_i}\}$ is the set of transmission tasks that are located at the channel c_j.
- The 4th term is the energy consumption rate caused by all assigned transmission tasks on d_i's sending end, where $e^{\mathrm{S}}_{c_j}$ denotes the energy consumption of sending one data unit through channel c_i, $n^{\mathrm{T}}_{t_k}$ denotes the number of transmitted data units per time unit of transmission task t_i, $\{c_j | (d_i, c_j) \in A_r\}$ is the set of channels that are leaving from device d_i, and $\{t_k | c_j \in \Phi(t_k), t_k \in K_{d_i}\}$ is the set of transmission tasks that are located at the channel c_j.

$$\begin{aligned} u(d_i) = \; & \sum_{t_k \in R_{d_i}} \min(n^{\mathrm{T}}_{t_k} \cdot \min\{e^{\mathrm{R}}_{c_j} | (c_j, d_i) \in A_r\}, n^{\mathrm{P}}_{t_k.parent} \cdot e^{\mathrm{op}}_{d_i}) \\ & + \sum_{t_k \in S_{d_i}} \min(n^{\mathrm{T}}_{t_k} \cdot \min\{e^{\mathrm{S}}_{c_j} | (d_i, c_j) \in A_r\}, n^{\mathrm{P}}_{t_k.child} \cdot e^{\mathrm{op}}_{d_i}) \end{aligned} \tag{5.2}$$

In Equation 5.2, the 1st term examines every unassigned transmission task t_k in R_{d_i} by (1) assigning it to a channel pointing to d_i that has the least required receiving energy consumption, i.e., $\min\{e^{\mathrm{R}}_{c_j} | (c_j, d_i) \in A_r\}$ or (2) assigning it together with its direct predecessor processing task ($t_k.parent$) to d_i. The minimum energy consumption rate of these two possibilities is the lower bound of the energy consumption rate on d_i caused by t_k. Therefore, the 1st term in Equation 5.2 gives the lower bound of the energy consumption rate on d_i caused by the entire set R_{d_i}. Similarly, the 2nd term in Equation 5.2 gives the lower bound of the energy consumption rate on d_i caused by the entire set S_{d_i}.

In every partial-assignment n, for a device d_i that has been assigned tasks already, $u(d_i)$ gives the lower bound estimation of the energy consumption rate caused by the unassigned tasks and $k(d_i)$ is the actual energy consumption rate caused by the already assigned tasks. The cost function $f(n)$ is thus calculated as follows:

$$f(n) = \frac{e^{\mathrm{TO}}_{d^*}}{k(d^*) + u(d^*)} \tag{5.3}$$

where d^* is the "bottleneck" device and it is defined as the device whose battery lifetime is minimal among all devices based on the current partial-assignment n. Hence, the value of $f(n)$ is the upper bound estimation of system lifetime of any full-assignment extended from n.

5.2.3 Experiments

To evaluate the effectiveness and efficiency of our proposed algorithm, we implemented a test environment in Java. It contains three modules. The first module is a task and resource DAG generator. It reads in the sizes of task and resource DAGs and the range of profiling information and generates a random problem, i.e., the topologies and labeling parameters of task and resource DAGs. The number of source vertices in the DAGs can be controlled and the number of sink vertices is a constant one. This decision is made based on the fact that most of telemonitoring applications have a single sink task, e.g., the epilepsy detection application in Figure 1.2. Thus, each problem class can be represented as $(|P|, |T|, |D|, |C|, |SP|, |SD|)$, where six parameters denote the number of processing tasks, transmission tasks, devices, channels, source processing tasks and source devices respectively. Fixed associations (capturing the local constraints) are set between source (sink) processing tasks to a randomly selected source (sink) device. The second module is a search order generator. It examines the task DAG generated by the first module and sorts the tasks according to a particular strategy, e.g., depth-first. The third module implements the proposed A*-based algorithm and reports a number of candidate task

assignments with their estimated performance, i.e., end-to-end delay, system battery lifetime and availability.

In this section, we report a set of experimental results from this test. All tests are performed on a PC running Windows XP with Intel Core(TM)2 CPU 2.4GHz and 2G RAM. For each user-defined problem class (($|P|, |T|, |D|, |C|, |SP|, |SD|$)) with randomly generated topology and profiling information, the results are averaged over 50 runs. All the measured algorithm completion time are in milliseconds.

Effectiveness

In order to understand how effective it is to reconfigure an MPMS based on an optimal task assignment, we tested our algorithm based on a context-aware reconfiguration scenario. In this scenario, we generate a random MPMS and configure it optimally for the system battery lifetime. After a simulated context change, the system performance is likely to degrade. Then we examine the benefit of adapting the system towards a new optimal configuration compared to a static approach that is no reconfiguration at all. The involved steps are as follows:

1. We randomly create a problem class (we tested (8,10,5,6,3,2) and (10,12,8,10,3,2)) and identify an optimal task assignment "α" with a maximum system battery lifetime $\Omega^{\text{li}}_{\text{current}}(\alpha)$. This represents an optimally configured MPMS.

2. We randomly select a channel resource "c_i" and multiple its energy consumption of sending one data unit ($e^{\text{S}}_{c_i}$) and receiving one data unit ($e^{\text{R}}_{c_i}$) with a predefined factor (named as context change ratio that is larger than 1). This is to simulate a change of power consumption rate for data transmission, e.g., increased power consumption for sending one bit of data due to an increase in distance or a handover from WLAN to GPRS.

3. Using this new context information, the system battery lifetime is calculated for "α" again. This new value is denoted as $\Omega^{\text{li}}_{\text{new}}(\alpha)$ and stands for the system battery lifetime after the context change if we do not change the system configuration. It can be proven easily that $\Omega^{\text{li}}_{\text{new}}(\alpha) \leq \Omega^{\text{li}}_{\text{current}}(\alpha)$. The equality holds true when the system battery life time is not determined by the device that "c_i" is pointing to or leaving from. Thus, the relative system battery lifetime decrease is: $\frac{\Omega^{\text{li}}_{\text{current}}(\alpha) - \Omega^{\text{li}}_{\text{new}}(\alpha)}{\Omega^{\text{li}}_{\text{current}}(\alpha)}$, e.g., the 3rd column in Table 5.1.

4. Using the proposed algorithm, we calculate an optimal task assignment "β" that has a maximal system battery lifetime ($\Omega^{\text{li}}_{\text{new}}(\beta)$) based on the new context information. Thus, the relative system battery lifetime decrease when the system configuration is dynamic, i.e., the system can be reconfigured according

Class ($\|P\|, \|T\|, \|D\|$,) ($\|C\|, \|SP\|, \|SD\|$)	Context change ratio	Lifetime decrease (static)	Lifetime (decrease) (dynamic)	Improved lifetime
(8,10,5,6,3,2)	2	10.8%	4.5%	9.4%
(8,10,5,6,3,2)	5	23.9%	12.9%	30.8%
(8,10,5,6,3,2)	10	40.2%	19.3%	92.5%
(8,10,5,6,3,2)	20	53.5%	18.4%	246.5%
(10,12,8,10,3,2)	2	7.6%	2.5%	7.1%
(10,12,8,10,3,2)	5	32.3%	14.5%	53.5%
(10,12,8,10,3,2)	10	43.0%	17.6%	109.2%
(10,12,8,10,3,2)	20	39.6%	7.5%	217.6%

Table 5.1: *The potential improvement on system battery lifetime resulted by dynamic system reconfiguration according to optimal task assignment*

to a new optimal task assignment, can be calculated as: $\frac{\Omega^{\text{li}}_{\text{current}}(\alpha)-\Omega^{\text{li}}_{\text{new}}(\beta)}{\Omega^{\text{li}}_{\text{current}}(\alpha)}$, e.g., the 4th column in Table 5.1.

5. We now can evaluate the effectiveness of our proposed algorithm by examining the improved system battery lifetime. Compared to a static approach, the improved battery lifetime of a dynamic approach is defined as: $\frac{\Omega^{\text{li}}_{\text{new}}(\beta)-\Omega^{\text{li}}_{\text{new}}(\alpha)}{\Omega^{\text{li}}_{\text{new}}(\alpha)}$, e.g., the 5th column in Table 5.1.

The experimental results are summarized in Table 5.1. For each problem class, we run the simulation with four different values of context change ratio, i.e., 2, 5, 10 and 20. These four values are realistic and reflecting the real world mobile devices. For example, the experiments performed on several commercially available PDAs in [RZ07] show that, for transferring the same amount data, the energy required by a GPRS connection is 10-30 times of the amount required by a WLAN connection. Our experiment clearly shows a significant improvement on system battery lifetime if the system can be reconfigured according to an optimal task assignment, sometimes more than tripling the system battery lifetime. We can also observe that the more significant the context change, the better improvement can be expected.

Impact of Task Ordering Strategy

Given one particular task DAG, a number of task search orders are eligible based on the general rules for ordering the tasks. For example, the two traditional strategies are depth-first and breadth-first as illustrated by examples in Section 5.1.2. Based on

some initial tests, we observed that the breadth-first strategy results in much worse average performance for all problem classes. Therefore, we omit further discussion about breadth-first search order in this thesis and do not recommend to use it in practice. Addition to the traditional approaches, we design three alternative task ordering strategies: heavy-first, depth/heavy-first and smallhop-first, and examine their impact on the search performance in this section.

- "Heavy-first": obeying the general rules, heavy-first sorts the tasks in the decreasing order based on their weighted number of operations ($n_{p_i}^{\mathrm{P}}$) and number of transmitted data units ($n_{t_i}^{\mathrm{T}}$). Thus, a task with "heavier" demand on processing or transmission will be assigned first. This is similar to the strategy applied in [RCD91, KA98], which ranks the tasks according to the total computation and communication cost.

- "Depth/Heavy-first": basically sorts tasks in a depth-first strategy and if several tasks are equal in their depth level, e.g., among source processing tasks or among direct successor transmission tasks leaving from the same processing task, the "heavier" task will be assigned first.

- "Smallhop-first": smallhop-first sorts tasks again in a depth-first strategy from source tasks to sink tasks. If several tasks are equal in their depth level, we further compare their distance, i.e., number of intermediate tasks, to the nearest sink processing task: the task with a smaller distance will be assigned first.

Two sets of experiments are carried out in order to examine the impact of different task search orders on the algorithm performance for various classes. In experiment "A", we tested the four identified ordering strategies on 9 problem classes. Classes are different from each other in terms of the number of processing tasks, transmission tasks, devices and channels $(|P|, |T|, |D|, |C|)$. They are: "0" (8, 10, 5, 6, 3, 2); "1" (9, 11, 5, 6, 3, 2); "2" (10, 12, 5, 6, 3, 2); "3" (8, 10, 6, 8, 3, 2); "4" (9, 11, 6, 8, 3, 2); "5" (10, 12, 6, 8, 3, 2); "6" (8, 10, 7, 9, 3, 2); "7" (9, 11, 7, 9, 3, 2); "8" (10, 12, 7, 9, 3, 2). For every class, the algorithm ran 50 times and the average search tree sizes resulting from these four orders are reported in Figure 5.4.

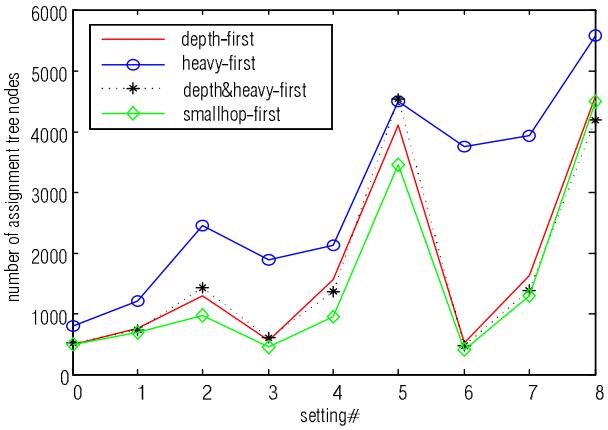

Figure 5.4: *Comparison of four ordering strategies by examining the generated search tree size in different settings (experiment "A")*

In experiment "B", we tested these four ordering strategies on 8 problem classes. Classes are different from each other in terms of the number of source processing tasks and source devices ($|SP|, |SD|$). They are: "0" (10, 12, 8, 10, 2, 2); "1" (10, 12, 8, 10, 3, 2); "2" (10, 12, 8, 10, 3, 3); "3" (10, 12, 8, 10, 4, 3); "4" (10, 12, 8, 10, 4, 4); "5" (10, 12, 8, 10, 5, 4); "6" (10, 12, 8, 10, 5, 5); "7" (10, 12, 8, 10, 6, 5). For every class, the algorithm ran 50 times and the average search tree sizes resulted by these four orders are reported in Figure 5.5.

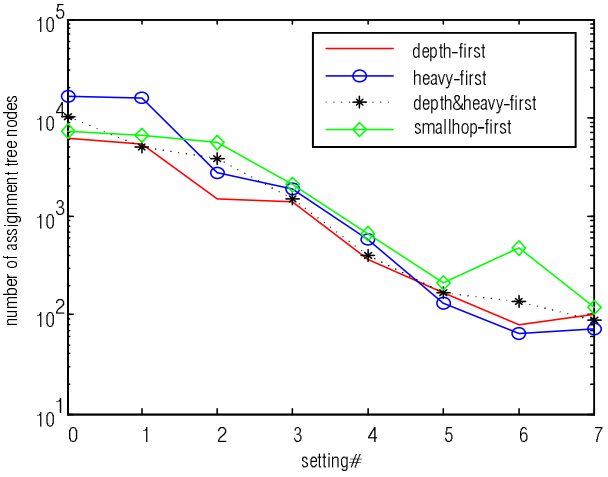

Figure 5.5: *Comparison of four ordering strategies by examining the generated search tree size in different settings (experiment "B")*

From both experiments, we observe that deliberately designed task search orders do not guarantee a better performance compared to a simple search order like depth-first. The reason lies in the complicated cost function and search tree construction. When a simpler cost function is used and no sophisticated assignment constraints are applied, e.g., as in the earlier research [RCD91, KA98], a search order that assigns tasks with a larger contribution on $f(n)$ first can lead to a better algorithm performance and it is relatively easy to identify such a search order. However, in our case, the joint influence of assignment constraint and the cost function formulation results no particular search order that can offer a significant better performance. Thus, we opt to use an easy and effective ordering strategy, i.e., depth-first, in the rest of this chapter.

Impact of Number of Candidate Assignments

Table 5.2 shows how the algorithm performs in computing a set of candidate assignments ($\{\Phi_m^{\text{can}}|m = 1, 2, N_{rc}\}$) with different required numbers (N_{rc}). The testing value of N_{rc} ranges from 1 to 5. The problem class is (8,10,6,8,4,3) and the results are averaged over 50 runs. From the experimental results, we made the following two observations:

1. Although the mean size of search tree and the mean completion time are not large for this problem class, their values are spread over a relatively large range. There are a few explanations. First, due to the randomly generated topology and the presence of the reachability constraint, some task and resource DAGs only have a limited number of possible assignments while some of them have many more possibilities. Second, due to the randomly generated profiling information, it is possible that some problem instances have very few possible task assignments fulfilling both the local constraint and the assurance constraint. Thus, this difference in the size of search space can vary a lot during the 50 runs.

2. Between searching for one best assignment (the optimal one) and searching for five candidate assignments, the difference regarding search tree size and algorithm completion time is not so significant. Thus, this is a very promising algorithm for computing a set of candidate assignments, as was our intention in the design of the MADE middleware (c.f. Section 3.2).

Bounded A*

Although the A*-based algorithm proves to be very effective (better than other admissible algorithms with the same heuristics) in finding candidate assignments, it

N_{rc}	♯Nodes	std	Time (ms)	std
1	3090.05	7147.64	167.65	643.07
2	3190.75	7143.19	168.00	533.18
3	3298.05	7133.26	168.10	526.03
4	3339.50	7140.35	165.75	517.24
5	3412.05	7136.94	179.80	529.77

Table 5.2: *Experiments on different number of required candidate assignments on the problem class of (8,10,6,8,4,3)*

still suffers from worst-case exponential complexity, i.e., the worst-case time and space complexity is at least $O(|D|^{|P|})$ where $|P|$ is the number of processing tasks and $|D|$ is the number of devices. For the A*-based algorithms to provide a reasonable solution in bounded time, it has been suggested to set an artificial size limit to the OPEN list [Rus92]: when the number of recorded partial-assignments exceeds this allowed maximum number, some still expandable partial-assignments whose upper bound battery lifetime are lower will be discarded by the algorithm, i.e., removed from the OPEN list. This bounded A*-algorithm is more "greedy" because it only traverses a number of branches with potentially higher battery lifetime. Four different size limits of OPEN list, i.e., $|P|\cdot|D|$, $0.8\cdot|P|\cdot|D|$, $0.5\cdot|P|\cdot|D|$ and $0.2\cdot|P|\cdot|D|$, are tested against running a normal (unbounded) A*-algorithm. For each test, we examine a so-called quality of assignment (QoA) measure:

$$QoA = \frac{mean(\Omega^{\text{li}}(\textit{candidate assignments found by bounded A*}))}{mean(\Omega^{\text{li}}(\textit{candidate assignments found by normal A*}))} \tag{5.4}$$

We tested the bounded A*-algorithm in three problem classes and the results are reported in Table 5.3, Table 5.4 and Table 5.5. The number of required candidate assignments (N_{rc}) is always set at 5. The 1st column in the table presents the size limit of OPEN list. The 2nd column presents the average number of nodes in the generated search tree. The 3rd column presents the standard deviation of search tree size. The 4th column depicts the search completion time and the 5th column indicates its standard deviation. The 6th column presents the number of found candidate assignments. All results are averaged over 50 runs with random graph structure and profiling information.

It is observed that even with a very tight bound, e.g., $0.2 \cdot |P| \cdot |D|$, the candidate assignments found by bounded A* are still satisfactory since they are on average worse than the optimal candidate assignments only within a 10% range. However, one additional drawback of the bounded A*-algorithm is that it does not always produce a sufficient number of candidate assignments. For example, 4.65 for the

case of "$0.2 \cdot |P| \cdot |D|$" in Table 5.3 and 4,85 for the case of "$|P| \cdot |D|$" in Table 5.4. The reason lies in its "greedy" nature, since some branches with valid full-assignments are removed from the (bounded) OPEN list and the left branches may not lead to a sufficient number of valid full-assignments.

Max size of $OPEN$	**♯Nodes**	**std**	**Time (ms)**	**std**	**♯Found**	**QoA**
unbounded	3497.05	7910.72	235.35	726.98	5.00	1
$\|P\| \cdot \|D\|$	777.75	754.79	16.25	12.93	5.00	98%
$0.8 \cdot \|P\| \cdot \|D\|$	721.25	688.74	16.45	12.85	5.00	97%
$0.5 \cdot \|P\| \cdot \|D\|$	570.30	563.98	12.60	16.61	5.00	95%
$0.2 \cdot \|P\| \cdot \|D\|$	283.00	195.40	7.00	7.95	4.65	93%

Table 5.3: *Experiments on bounded A* on the problem class of (8,10,6,8,4,3)*

Max size of $OPEN$	**♯Nodes**	**std**	**Time (ms)**	**std**	**♯Found**	**QoA**
unbounded	1265.55	2124.13	40.65	74.42	5.00	1
$\|P\| \cdot \|D\|$	471.30	311.22	16.40	20.03	4.85	97%
$0.8 \cdot \|P\| \cdot \|D\|$	433.70	275.74	11.60	6.89	4.80	97%
$0.5 \cdot \|P\| \cdot \|D\|$	338.65	190.16	7.80	8.01	4.85	96%
$0.2 \cdot \|P\| \cdot \|D\|$	226.65	109.42	5.60	7.83	5.00	90%

Table 5.4: *Experiments on bounded A* on the problem class of (7,9,6,8,3,2)*

Max size of $OPEN$	**♯Nodes**	**std**	**Time (ms)**	**std**	**♯Found**	**QoA**
unbounded	757.90	631.51	25.05	24.60	5.00	1
$\|P\| \cdot \|D\|$	484.10	287.63	15.65	16.78	5.00	99%
$0.8 \cdot \|P\| \cdot \|D\|$	458.00	266.22	9.35	7.84	5.00	98%
$0.5 \cdot \|P\| \cdot \|D\|$	357.95	236.36	8.60	9.53	5.00	96%
$0.2 \cdot \|P\| \cdot \|D\|$	183.60	63.87	4.60	7.21	4.95	89%

Table 5.5: *Experiments on bounded A* on the problem class of (7,9,5,7,3,2)*

5.3 Minimize End-to-End Delay

This section proposes a different variant of A*-based task assignment algorithm. This algorithm takes the end-to-end delay as the objective function and considers the other two measures in the assurance constraint. Similar to the previous section, we present the algorithm design first and finish with performance evaluations.

5.3.1 Formulation

We formulate the task assignment problem of minimizing end-to-end delay as follows:

- **Given**: (1) a telemonitoring application (P, T, A_t, L_P, L_T); (2) an m-health platform (D, C, A_r, L_D, L_C); (3) Three performance evaluation function Ω^{de}, Ω^{li} and Ω^{av} for task assignments that represent battery lifetime, end-to-end delay, and availability.
- **Goal**: To find a number of candidate task assignments $\{\Phi_m^{\mathrm{can}} | m = 1, 2, ..., N_{rc}\}$ among all possible task assignments for which the values of $\Omega^{\mathrm{de}}(\Phi_m^{\mathrm{can}})$ are minimized.
- **Subject to**: four assignment constraints namely *type constraint*, *local constraint*, *assurance constraint* and *reachability constraint*.

The *assurance constraint* defined here is slightly different with the previous general definition (c.f. Section 3.3.3) by removing the maximum allowed end-to-end delay. The other three assignment constraints are the same as in the general definition. The high-level routine of the A*-based task assignment algorithm with the objective of minimizing the system end-to-end delay is illustrated in Algorithm 5.

5.3.2 Function $f(n)$

The end-to-end delay of a given assignment is defined as the maximum of all task paths' end-to-end delays (c.f. Section 3.3.3). The value of this cost function increases monotonically or remains the same when the search tree is expanding. To benefit from the design idea of A*-algorithm, we introduce the function $f(n)$ in this particular problem as the lower-bound estimation of end-to-end delay of all full-assignments extended from a partial-assignment n. The search should always expand the search tree from a partial-assignment with the lowest $f(n)$.

$$f(n) = max\{g(y) + h(y) | y \in Y\} \tag{5.5}$$

Algorithm 5 *Pseudo code of A*-based task assignment algorithm for minimizing the end-to-end delay (The grey highlighted parts are different with the earlier Algorithm 4)*

1: input the number of required candidate assignments as N_{rc}
2: input the minimal required availability $min\Omega^{\text{av}}$ and the minimal required system battery lifetime $min\Omega^{\text{li}}$
3: determine a search order for the tasks: $order$
4: initialize a sorted list $OPEN$ to contain the already visited partial-assignments
5: initialize a sorted list $candidates$ to contain the already found candidate assignments
6: create a partial-assignment $root$, $f(root) := 0$, $root.toBeAssignedTask := order.first$ /* For a partial-assignment n, $n.toBeAssignedTask$ contains the next to be assigned task according to $order$*/
7: add $root$ into $OPEN$
8: **while** true **do**
9: **if** $OPEN$ is empty **then**
10: **return** $candidates$ /* Found candidate assignments are less than the required number N_{rc}*/
11: **end if**
12: remove a partial-assignment n with lowest $f(n)$ from $OPEN$ /* Currently, the most promising partial-assignment */
13: **if** n is a full-assignment **then**
14: add n into $candidates$ /* Found a new candidate */
15: **if** $candidates.size() = N_{rc}$ **then**
16: **return** $candidates$ /* A list of N_{rc} found candidate assignments */
17: **end if**
18: **end if**
19: expand from n to a set of child partial-assignments $\{cn_i\}$ by assigning $n.toBeAssignedTask$
20: $index := order.getIndex(n.toBeAssignedTask)$ /* Record the index of $n.toBeAssignedTask$ */
21: **for all** partial-assignments in $\{cn_i\}$ **do**
22: **if** no data dependency violation at cn_i **then**
23: **if** $\Omega^{\text{li}}(cn_i) \geq min\Omega^{\text{li}} \wedge \Omega^{\text{av}}(cn_i) \geq min\Omega^{\text{av}}$ **then**
24: /* cn_i is a valid partial-assignment, we should record it in $OPEN$ */
25: calculate $f(cn_i)$
26: $cn_i.toBeAssignedTask := order.get(index + 1)$ /* Record the next to be assigned task */
27: add cn_i into $OPEN$
28: **end if**
29: **end if**
30: **end for**
31: **end while**

where Y is the set of tasks that (1) are assigned already in a partial-assignment n and (2) still have unassigned direct successor tasks, $g(y)$ is the maximum end-to-end delays between a source processing task and the task y, $h(y)$ is the lower-bound estimation of the additional delay caused by all unassigned successor tasks of the task y.

To facilitate a more efficient calculation of $g(y)$, we use a table (named as $ExactDelayOfTask$) to store the maximum end-to-end delays between a source processing task and each assigned task. Thus, when assigning a new task, we can calculate the maximum end-to-end delay of this task based on the stored delay information of its direct predecessor tasks. In this way, each time when a new task is assigned, we avoid calculating the end-to-end delay for the entire task path connecting a source processing task and this newly assigned task.

To guarantee $h(y)$ is a lower-bound estimation on end-to-end delay for all full-assignments, $h(y)$ is defined as the following:

$$\begin{aligned} h(y) \quad = \quad & \max\{\sum_{m=1}^{|tp_j^y|}(\min_{tp_j^y(m)\in P}(de^{\mathrm{P}}_{tp_j^y(m),*}) \\ & + \min_{tp_j^y(m)\in T}(de^{\mathrm{T}}_{tp_j^y(m),*}))|tp_j^y \in TP^y, j = 1,2,...|TP^y|\} \end{aligned} \tag{5.6}$$

where TP^y is the set of task paths $\{tp_j^y\}$ in the task DAG that connect a direct successor task of y with a sink (processing) task, "*" denotes any device or channel resource, and m is the index of task in a task path. For each tp_j^y, we calculate its minimal end-to-end delay. The maximum value among them determines the value of $h(y)$.

We propose a labeling procedure to calculate $h(y)$ for each task y in a task DAG. This is a pre-processing procedure that is executed before the search. The labeling starts from sink tasks and propagates reversely along the directions in a task DAG. All the sink tasks are labeled with "0" initially.

- For a processing task y, we calculate its $h(y)$ as follows:

$$h(y) = \max_{(y,x)\in A_t}\left\{h(x) + \min\{de^{\mathrm{T}}_{x,*}\}\right\} \tag{5.7}$$

  where x denotes a direct successor transmission task of y, "*" denotes any device or channel resource.

- For a transmission task y, we calculate its $h(y)$ as follows:

$$h(y) = h(x) + \min\{de^{\mathrm{P}}_{x,*}\} \tag{5.8}$$

  where x denotes the direct successor processing task of y, "*" denotes any device resource.

This labeling procedure visits every task exactly once. For each task, we examine its processing or transmission delay at most once per device or channel resource. Thus, the time complexity of this labeling procedure is $O\left((|P|+|T|)(|D|+|C|)\right)$.

5.3.3 Semi-Topological Task Search Order

A possible depth-first task search order for the task DAG shown in Figure 5.6 is: $\{p_1$, t_1, p_2, t_2, p_3, t_3, p_4, t_4, p_7, t_5, p_5, t_6, p_5, t_7, p_6, $t_8\}$. Consider for a moment how we calculate $g(n)$, in particular the use of $ExactDelayOfTask$, we see a potential inefficiency caused by backtracking. Following this order, when "p_4" is assigned, the maximum delay occurred at "p_4" is calculated only based on the task path connecting "p_1" and "p_4" via "t_3" and this result is used further to calculate for the successor tasks of "p_4", e.g., "t_4". However, it is possible that the maximum end-to-end delay between a source processing task and "p_4" is determined by the task path via "t_6". If this happens, some additional and complicated updates have to be performed upon the existing entries in the $ExactDelayOfTask$ table at the partial-assignment node of assigning "t_4".

A solution to avoid backtracking is to design an ordering strategy by taking into consideration how the $ExactDelayOfTask$ table is used. We name this ordering strategy as "semi-topological". It is very similar to a topological sort of DAG. The only difference is that we have to obey the general rules for ordering tasks (c.f. Section 5.1.2): if a transmission task (α) is visited, then its direct successor (processing) task (β) should be visited immediately. This is even the case when not all direct predecessor (transmission) tasks of β are visited, i.e., a violation to topological sorts. For example, a task order generated following this "semi-topological" approach is illustrated in Figure 5.6. Based on this new "semi-topological" search order, task "t_4" will not be assigned until all its predecessor tasks have been assigned already.

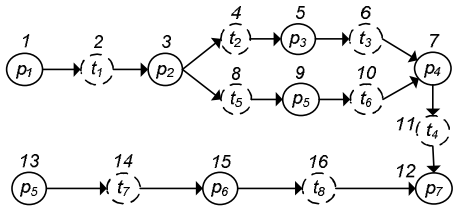

Figure 5.6: *A "semi-topological" task order for an example task DAG. Different from a topological sort, p_4 comes before t_6 and p_7 comes before t_8*

5.3.4 Experiments

To evaluate the effectiveness and efficiency of this new variant of A*-algorithm, we performed a set of experiments in a Java implementation. The experiment setting is the same as explained in Section 5.2.3.

Effectiveness on Minimizing End-to-End Delay

We first conducted an effectiveness test to understand better what is the added-value on end-to-end delay by enforcing an optimal task assignment in MPMS. Six different problem classes were tested and the simulation results are presented in Table 5.6. In the test of each problem class, we ran the A*-based algorithm to compute the optimal task assignment with a minimal delay ($\Omega^{\mathrm{de}}_{\mathrm{opt}}$). As a comparison, we also computed the average end-to-end delay ($\Omega^{\mathrm{de}}_{\mathrm{ave}}$) of all possible task assignments. The relative improvement on end-to-end delay by enforcing an optimal task assignment is defined as $\frac{\Omega^{\mathrm{de}}_{\mathrm{ave}}-\Omega^{\mathrm{de}}_{\mathrm{opt}}}{\Omega^{\mathrm{de}}_{\mathrm{ave}}}$. The results show that the possible improvement we can expect for an optimal task assignment is a bit more than 20%.

Class	Delay (optimal)	std (optimal)	Delay (average)	std (average)	Improvement
(7,9,6,8,3,2)	336.86	66.45	484.32	51.98	30%
(7,9,6,8,4,3)	302.88	49.08	379.99	47.14	20%
(8,10,6,8,3,2)	368.74	53.48	503.44	57.55	26%
(8,10,6,8,4,3)	313.02	46.03	399.05	51.57	21%
(9,11,6,8,3,2)	398.52	60.45	531.16	62.40	25%
(9,11,6,8,4,3)	367.12	58.37	449.56	51.55	18%

Table 5.6: *The difference on end-to-end delay between optimal assignment and random assignments*

Impact of Number of Candidate Assignments

We tested the algorithm performance on computing candidate task assignments ($\{\Phi^{\mathrm{can}}_m | m = 1, 2, ...N_{rc}\}$) in different required number (N_{rc}). The testing value of N_{rc} ranges from 1 to 5. In each test, we measured the number of assignment nodes visited by the A*-based algorithm and the algorithm completion time. The simulation results for two problem classes are reported in Table 5.7 and Table 5.8. Our observation here is similar to the one made in the previous experiment on A*-based algorithm for maximizing system battery lifetime:

- The number of visited assignment nodes (search tree size) and the completion time are spread over a large range.
- Between searching for one best assignment (the optimal one) and searching for five candidate assignments, the difference regarding tree size and completion time is not so significant. Thus, this is a suitable algorithm for computing a small set of candidate task assignments.

N_{rc}	♯Nodes (mean)	♯Nodes (std)	Time (mean)	Time (std)
1	941.88	1080.15	28.36	38.12
2	971.50	1072.61	29.74	33.88
3	987.84	1075.64	31.52	35.41
4	1013.20	1100.14	30.60	37.58
5	1031.68	1108.20	31.32	37.34

Table 5.7: *Experiments on different N_{rc} on the problem class of (8,10,6,8,4,3)*

N_{rc}	♯Nodes (mean)	♯Nodes (std)	Time (mean)	Time (std)
1	8560.20	12103.29	563.08	1202.34
2	8579.86	12095.07	536.62	1174.70
3	8714.60	12113.06	540.00	1176.44
4	8791.56	12115.11	552.48	1178.90
5	9183.00	12199.72	575.26	1183.28

Table 5.8: *Experiments on different N_{rc} on the problem class of (9,11,6,8,3,2)*

Bounded A*

Here, we set size limit to $OPEN$ list and tested the bounded A*-algorithm. Four different size limits of $OPEN$ list, i.e., $|P| * |D|$, $0.8 * |P| * |D|$, $0.5 * |P| * |D|$ and $0.2 * |P| * |D|$, are tested against running an unbounded A*-algorithm. For each different size limit, we examined the quality of assignment (QoA) that is defined as:

$$QoA = \frac{mean\{\Omega^{\mathrm{de}}(\Phi_m^{\mathrm{can}} \textit{ found by bounded A*})\}}{mean\{\Omega^{\mathrm{de}}(\Phi_m^{\mathrm{can}} \textit{ found by unbounded A*})\}} \tag{5.9}$$

Thus, a value of QoA closer to "1" indicates that the found candidate assignment's end-to-end delay are closer to the optimal candidate assignments. We experimented this bounded A*-algorithm in three problem classes. The results averaged

over 50 runs are reported in Table 5.9, Table 5.10 and Table 5.11. It is observed that with a reasonable tight bound, e.g., $0.8 \cdot |P| \cdot |D|$, the candidate assignments found by bounded A* are still satisfactory since they are on average worse than the optimal candidate assignments only within a 5% range.

Max size of $OPEN$	**♯Nodes (mean)**	**♯Nodes (std)**	**Time (mean)**	**Time (std)**	**♯Found**	**QoA**
unbounded	2440.85	1196.85	77.55	44.20	5.00	1
$\|P\| \cdot \|D\|$	1102.70	309.08	24.15	17.98	5.00	1.04
$0.8 \cdot \|P\| \cdot \|D\|$	966.20	279.18	21.05	7.50	5.00	1.04
$0.5 \cdot \|P\| \cdot \|D\|$	564.90	175.61	14.10	4.84	5.00	1.07
$0.2 \cdot \|P\| \cdot \|D\|$	270.45	52.13	3.15	6.47	4.95	1.17

Table 5.9: *Experiments on bounded A* on the problem class of (8,10,6,8,3,2)*

Max size of $OPEN$	**♯Nodes (mean)**	**♯Nodes (std)**	**Time (mean)**	**Time (std)**	**♯Found**	**QoA**
unbounded	1085.85	1374.57	35.77	57.73	5.00	1.00
$\|P\| \cdot \|D\|$	674.96	555.54	16.70	16.45	5.00	1.03
$0.8 \cdot \|P\| \cdot \|D\|$	599.96	413.62	14.00	9.85	5.00	1.03
$0.5 \cdot \|P\| \cdot \|D\|$	452.55	312.24	10.04	8.95	5.00	1.04
$0.2 \cdot \|P\| \cdot \|D\|$	223.91	127.66	7.34	7.92	4.72	1.08

Table 5.10: *Experiments on bounded A* on the problem class of (8,10,6,8,4,3)*

Max size of $OPEN$	**♯Nodes (mean)**	**♯Nodes (std)**	**Time (mean)**	**Time (std)**	**♯Found**	**QoA**
unbounded	3577.80	3709.58	157.80	236.93	5.00	1.00
$\|P\| \cdot \|D\|$	1675.15	906.77	31.95	19.24	5.00	1.02
$0.8 \cdot \|P\| \cdot \|D\|$	1527.60	802.96	28.25	10.90	5.00	1.03
$0.5 \cdot \|P\| \cdot \|D\|$	975.05	519.21	18.65	11.96	5.00	1.04
$0.2 \cdot \|P\| \cdot \|D\|$	395.85	162.44	9.45	7.92	5.00	1.10

Table 5.11: *Experiments on bounded A* on the problem class of (9,11,6,8,4,3)*

Advantage of A* over Pure Greedy Algorithms

Since the task assignment problem in general form is an NP-hard problem, different heuristic algorithms have been proposed [AAH05, ZWS+08]. These algorithms compute one sub-optimal task assignment by greedily assigning a task to a device with a minimal assignment cost. We implemented an algorithm called "Pure Greedy" based on this heuristic approach and tested its performance against our A*-based algorithm. We conducted the experiments of finding one sub-optimal task assignment in two problem classes as shown in Table 5.12 and Table 5.13. We examined two measures. The first is a "success ratio" that is the probability of finding one (sub-)optimal assignment successfully. The second is "quality of the best assignment" (QoB) that is defined as:

$$QoB = \frac{\Omega^{\mathrm{de}}(\textit{sub-optimal assignment found})}{\Omega^{\mathrm{de}}(\textit{optimal assignment})} \tag{5.10}$$

From the experiment results, we see that although it is very fast, the "Pure Greedy" algorithm provides much worse results in both classes. In about 40 out of 50 runs, it can not find a possible task assignment. The reason is that this heuristic algorithm is very shortsighted and does not work well in our formulated task assignment problem. Our specific assignment constraints are the main cause of the unsuccessful use of the "Pure Greedy" algorithm: It often runs into an invalid assignment that either violates a local constraint, e.g., exceeds the available bandwidth on a particular channel, or violates a reachability constraint, e.g., two connected tasks are not assigned to two connected resources.

Max size of $OPEN$	**♯Nodes (mean)**	**♯Nodes (std)**	**Time (mean)**	**Time (std)**	**Success ratio**	**QoB**
unbounded	7192.15	13816.39	559.45	1614.08	100%	1
$\|P\| * \|D\|$	1122.25	837.13	23.35	23.60	100%	1.02
$0.8 * \|P\| * \|D\|$	966.65	652.91	20.25	11.39	100%	1.02
$0.5 * \|P\| * \|D\|$	712.45	433.29	14.75	9.48	100%	1.02
$0.2 * \|P\| * \|D\|$	381.65	192.07	4.70	7.37	100%	1.05
Pure Greedy	9.60	4.81	3.95	7.02	16%	1.14

Table 5.12: *Comparison the bounded A* with pure greedy algorithm on the problem class of (10,12,6,8,4,3)*

Max size of $OPEN$	♯Nodes (mean)	♯Nodes (std)	Time (mean)	Time (std)	Success ratio	QoB
unbounded	3151.60	3347.59	120.30	168.25	100%	1
$\|P\| * \|D\|$	1127.90	801.76	25.05	17.90	100%	1.02
$0.8 * \|P\| * \|D\|$	912.05	540.47	17.85	11.55	100%	1.02
$0.5 * \|P\| * \|D\|$	602.10	370.99	12.60	9.62	100%	1.03
$0.2 * \|P\| * \|D\|$	324.60	196.67	8.65	8.03	100%	1.07
Pure Greedy	10.65	4.03	3.80	8.42	22%	1.17

Table 5.13: *Comparison the bounded A* with pure greedy algorithm on the problem class of (9,11,6,8,4,3)*

5.4 Concluding Remarks

This chapter presents A*-based task assignment algorithms for MPMSs. Although the general form of the problem, i.e., DAG-DAG, is NP-hard, we expect the A*-algorithm to be a suitable solution when the problem size is small. For example, the epilepsy detection scenario presented in Section 1.1.2 deals with 10 processing tasks and less than 10 devices. Our choice is further motivated by the fact that the A*-algorithm is admissible and guarantees a better performance, e.g., generating a smaller search tree, than any other admissible search algorithms [DP85].

In Section 5.1, we described several general design aspects of the A*-based algorithms that are independent of the objective function. First, we illustrated what each node in the search tree represents and how to encode task assignments into the search tree. Second, we explained the concept of task search order and its impact on the algorithm performance. Third, we proposed two particular designs, namely node expansion rules and data dependence check, to ensure the traversed search tree to be both effective and efficient.

After presenting the shared design aspects, we studied two variants of A*-based task assignment algorithms in Section 5.2 and Section 5.3 with different objective function. The performance of both algorithms are evaluated based on Java implementations. The experiments proved that the A*-based task assignment algorithm is a suitable candidate for the task assignment problem in MPMSs when the number of involved tasks and devices are limited. In case we have to deal with a larger problem, the bounded version of the A*-based algorithm is shown to be an effective (in terms of the quality of the solution) and efficient (in terms of the computational resource required) alternative solution.

Chapter 6
Task Distribution Infrastructure

The proposed MADE middleware provides four functionalities: *Monitoring*, *Analysis*, *Decision* and *Enforcement*. In Chapter 3, the high-level design of this middleware layer has been presented. Chapter 4 and Chapter 5 are dedicated to task assignment algorithms that provide the intelligence of the Analysis functionality. This chapter is, in fact, a more in-depth study on the Monitoring, Decision and Enforcement functionalities of the MADE middleware. This chapter presents a task distribution infrastructure that supports the dynamic reconfiguration and a computational model to evaluate the reconfiguration cost. In this thesis, the reconfiguration cost is defined in terms of reconfiguration time.

This chapter is structured as follows. Section 6.1 presents the architecture and behavior of the task distribution infrastructure. Section 6.2 defines the concept of affected tasks which form the targeted area of a system reconfiguration. Section 6.3 introduces a 7-step scheme to construct reconfiguration plans. Section 6.4 presents a computational model to quantitatively evaluate reconfiguration costs. Section 6.5 presents a proof-of-concept implementation of the proposed task distribution infrastructure and reports its performance.

6.1 Architecture and Behavior

To support dynamic task distribution, we propose an infrastructure that supports task distribution based system reconfigurations as depicted in Figure 6.1. It consists of a set of Bio-Signal Processing Units (BSPUs), a set of Facilitators, and a Coordinator. A "Facilitator" is deployed on each device in an MPMS and represents its hosting device in MADE. A Coordinator is a central component that computes optimal task assignments and controls the Facilitators to deploy a target task assignment by means of task distribution. BSPU is the unit of distributable tasks. This infrastructure is partially inspired by Service Oriented Architecture (SOA), where a BSPU can be viewed as a service implementing bio-signal processing tasks, Facilitators act like service providers and the Coordinator acts like a service registry. As shown in Figure 6.1, when a new device joins the MPMS or its context information changes,

its hosted Facilitator informs the Coordinator about its presence and current context information (Interaction 1). Once a decision to reconfigure the MPMS is made by the Coordinator, it controls each Facilitator to reconfigure its hosting device (Interaction 2). After the reconfiguration is finished successfully, the new telemonitoring application will be formed by a set of distributed BSPUs (Interaction 3). In the following part of this section, we explain each architectural component in more detail. We use

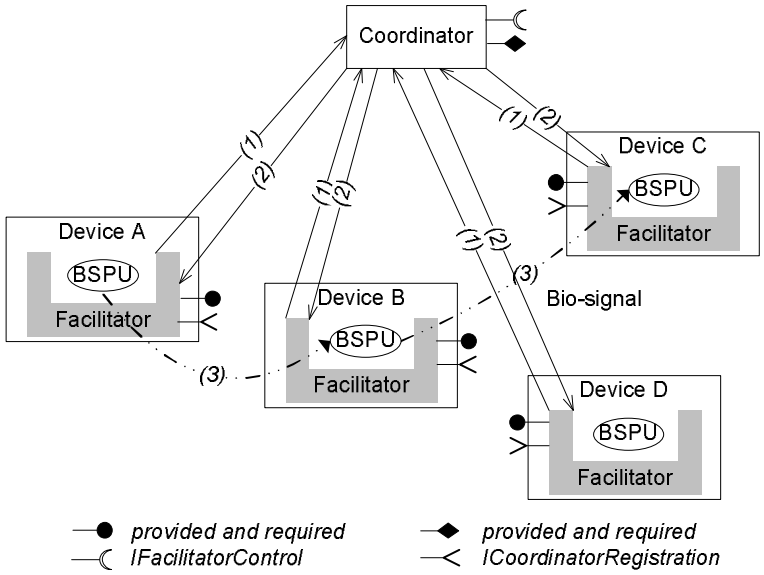

Figure 6.1: *The system architecture of task distribution infrastructure.*

OSGi technology[1] as the implementation framework for the task distribution infrastructure and Java as the programming language. These two technology choices are motivated for the following reasons:

1. They are both designed for cross-platform systems which are the case for many MPMSs.
2. They are both well supported by standardization bodies and the industry.
3. They are both very suitable for resource-constrained devices: Java Platform, Micro Edition (formerly J2ME) has been integrated on many mobile and embedded devices. The OSGi specification originates from the consumer electronic industry with the aim of providing a light-weight device management solution.

[1]http://www.osgi.org

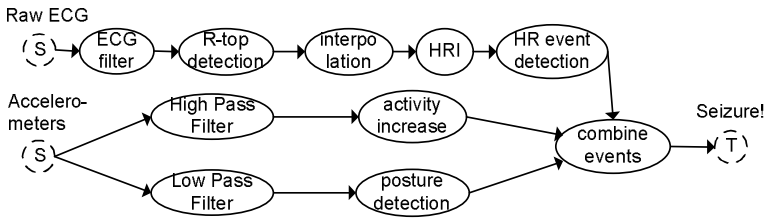

Figure 6.2: *A seizure detection algorithm (a copy of Figure 1.2)*

4. Existing OSGi implementations often offer basic remote management functionalities which greatly reduce our development effort in the task distribution infrastructure.

6.1.1 BSPU - A Unit of Composition

To realize the distributed execution of telemonitoring applications and to enable task distribution, we introduce a software component named as BSPU which plays the role of a unit of system composition. A BSPU receives continuous bio-signal streams via its input buffer from its predecessor BSPUs, manipulates the streams, and transfers the processed output streams to the successor BSPUs if any. The tasks shown in Figure 6.2 can all be implemented as BSPUs: for example, a BSPU can implement the task of "R-top detection" which takes the filtered ECG signal as input and outputs all detected "R peak" in the ECG signal.

Besides the data plane functionality, a BSPU also provides a set of control methods so that an external entity can manage its lifecycle. Thus, the task distribution and dynamic reconfiguration can be achieved. With the choice of the OSGi technology, the BSPU lifecycle model is designed by extending the OSGi bundle lifecycle (Figure 6.3). Similar to an OSGi bundle, a BSPU can be installed, uninstalled, started, stopped and updated. When a BSPU is installed, its code is loaded into the OSGi framework on a target device. The framework then further checks this BSPU's software dependencies and tries to install and start its required service packages. After these resolvements, a BSPU can be uninstalled, updated or started. An "uninstall" call removes a BSPU from the framework. An "update" call installs a new version of the BSPU into the OSGi framework. A "start" call activates the bio-signal processing task implemented by this BSPU. When a BSPU is stopped, its provided services will be eliminated, but its installed code still resides in the OSGi framework.

In the dynamic reconfiguration of an MPMS, 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, the proposed

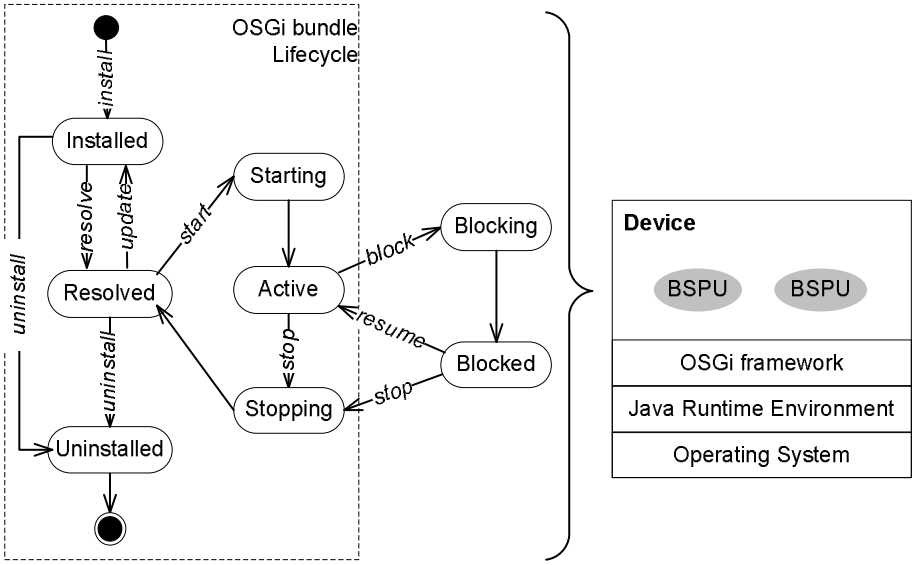

Figure 6.3: *Lifecycle of a BSPU which is extended based the component lifecycle in OSGi framework*

task distribution infrastructure should support execution states transfer, i.e., stateful reconfiguration. For this purpose, we introduce two new states and two new transitions that can be used to temporarily freeze a set of BSPUs and allow a stateful reconfiguration. A "block" call enforces a transition to the "Blocking" state. In this state the BSPU will no longer take and process incoming data streams from its input buffer but just finishes the ongoing processes. After it finishes all the ongoing processes, this BSPU will move to the "Blocked" state where it no longer sends any data streams to its successor BSPUs. The last message sent from a BSPU to its successor BSPUs before it moves into the "Blocked" state contains a "Blocked" flag. This flag propagates through all its successor BSPUs to inform them about the applied "block" action. After a BSPU receives a "Blocked" flag, it enters a special "Active" state, named as "Quiescent". In the "Quiescent" state, a BSPU does not actively perform bio-signal processing tasks anymore since there is no sufficient new input signal. This is because that, as we stated earlier in our assumption (Section 3.3.1), a task can only function when the bio-signals from all its predecessor tasks are received and synchronized. A "resume" call unblocks a "Blocked" BSPU and enables it to fetch the received data in its input buffer and process incoming data streams again. When a BSPU is in the state "Active" or "Blocked", it is possible to inspect or configure the execution state information.

We distinguish two types of BSPU: processing and relaying. Each processing

BSPU implements a processing task, e.g., "p_2" in Figure 3.4. *In this thesis, we use the same notation of "p_i" to represent a processing task or a processing BSPU. When it occurs, the exact meaning should be clear from the surrounding text.* Relaying BSPUs do not contain any data manipulation functions and are implemented to support transmission tasks. We denote the relaying BSPU of a transmission task "t_i" as "t_i*". For example, in the task assignment "α" as defined in Table 6.1, "t_2" is assigned to "d_1, c_1, d_2, c_2, d_5". Hence, one "t_2*" should be deployed onto device "d_2" to support the execution of "t_2".

6.1.2 Facilitator

Facilitator is the representative of its hosting device in the task distribution infrastructure: it declares the presence of a device, reports the local device context information to the Coordinator, receives control commands from the Coordinator to dynamically configure its locally hosted BSPUs. To enable the Coordinator to have control over Facilitators, each Facilitator provides a *Facilitator Control* interface as shown in Figure 6.1. This interface provides the following methods (Figure 6.4):

- *getAllBSPU*: to retrieve a list of all hosted BSPUs at this Facilitator.
- *installBSPU*: to install a new BSPU at the Facilitator based on the given source.
- *uninstallBSPU*: to delete the implementation of a BSPU and release its occupied resource, e.g., memory space.
- *updateBSPU*: to update the implementation of an inactive BSPU. It can be viewed as a composition of "uninstallBSPU" and "installBSPU".
- *startBSPU*: to start the bio-signal processing task implemented by the selected BSPU.
- *stopBSPU*: to stop the bio-signal processing task implemented by the selected BSPU.
- *configureBSPU*: to configure the execution parameters of a BSPU.
- *inspectBSPU*: to retrieve a list of all configurable execution parameters of a BSPU
- *blockBSPU*: to stop a BSPU from fetching and processing new incoming data streams from its input buffer and just let it finish the ongoing processes.
- *resumeBSPU*: to enable a BSPU to start fetching and processing new incoming data streams from its input buffer.

- *isBSPU*: to start a composite action consisting of "installBSPU" and "startBSPU".
- *iscBSPU*: to start a composite action consisting of "installBSPU", "startBSPU", and "configureBSPU".
- *suBSPU*: to start a composite action consisting of "stopBSPU" and "uninstallBSPU".

The first six methods in the above list are similar to what the OSGi framework supports already. The rest are defined and implemented to offer the additional functionalities required by the task distribution infrastructure.

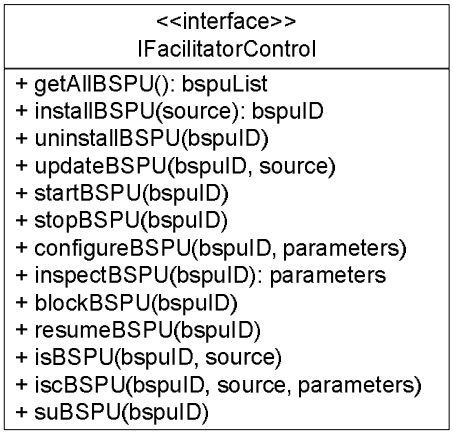

Figure 6.4: *The interface of Facilitator Control. "bspuID" is a unique identifier. "source" is either the source installation file of a BSPU or a pointer to the installation file. "parameters" is a list of* $< variable, value >$ *pairs which represent configurable execution state information of a BSPU*

.

6.1.3 Coordinator

Coordinator is the architectural component that hosts task assignment algorithms. Besides providing the necessary intelligence of identifying the optimal system configuration, the Coordinator also supports system reconfigurations by means of managing available Facilitators and coordinating the system reconfiguration actions. As depicted in Figure 6.5, a Coordinator consists of the following six modules:

1. "Facilitator registration/update": This module maintains a "Coordinator Registration" interface that can be invoked by external entities to inform the Coordinator about new Facilitators and context changes. This interface will be detailed in the later part of this section.

2. "Facilitators registry": This is a registry to store the information about available Facilitators and their context information, e.g., hosted BSPUs and network connectivity.

3. "Required telemonitoring application": This is the module providing the information about required telemonitoring applications, i.e., the labeled "task DAG".

4. "Task assignment algorithm": This module holds the task assignment algorithms proposed in the earlier chapters.

5. "Reconfiguration plan generator": This module takes a computed target task assignment as the input, identifies the affected tasks and computes a reconfiguration plan. We explain affected tasks in Section 6.2 and how to generate a reconfiguration plan in Section 6.3.

6. "Control commands dispatcher": This module coordinates the appropriate Facilitators (through their Facilitator Control interfaces) to reconfigure an MPMS according to a reconfiguration plan.

After depicting the structural view of Coordinator in Figure 6.5, we explain the behavior of Coordinator by presenting its state diagram as shown in Figure 6.6. The Coordinator operates in four distinguished states, i.e., "Monitoring", "Decision-making", "Updating (facilitators registry)", and "Dispatching (commands)". These four states are closely related with the four main functionalities of the MADE middleware: *Monitoring*, *Analysis*, *Decision* and *Enforcement*.

1. In the "Monitoring" state, the Coordinator monitors the context changes in an MPMS by receiving reports from external entities, e.g., Facilitators.

2. In the "Decision-making" state, the Coordinator performs the analysis and decision functionality.

3. In the "Updating" state, the Coordinator performs maintenance on the "Facilitators registry", which is a bookkeeping activity to support the four main functionalities.

4. In the "Dispatching" state, the Coordinator performs the enforcement functionality to dynamically reconfigure an MPMS.

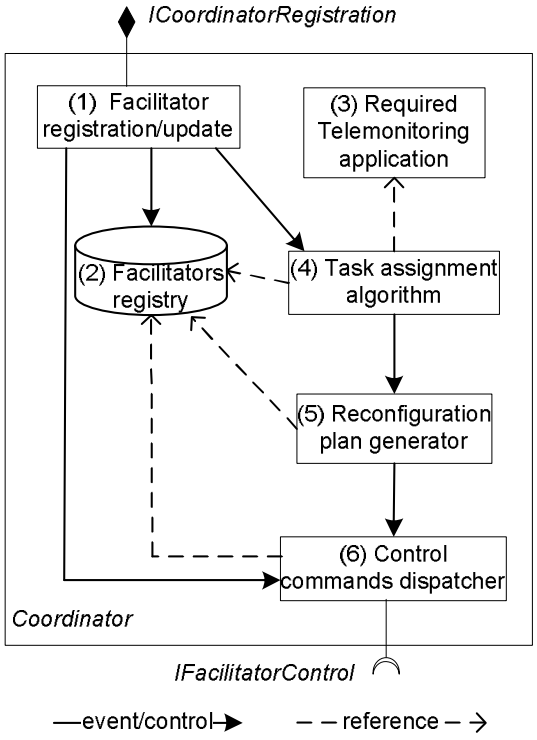

Figure 6.5: *The internal architecture of Coordinator*

The provided Coordinator Registration interface of a Coordinator (Figure 6.7) consists of the following methods:

- *registerFacilitator*: This method is used to register a new Facilitator at the Coordinator. This happens when a device joins the MPMS and becomes ready for performing bio-signal processing tasks.
- *updateFacilitator*: This method updates the context information of a registered Facilitator at the Coordinator. The updated context information, i.e., "fContext", can be a network connection change, CPU load change, the fact that the remaining battery energy reduces to a certain threshold value, etc. The study on detailed representation format of "fContext", so called modeling of context, is not the focus of this thesis. However, there exist many works on this topic and interested readers are referred to [CC03, Cos07].
- *unregisterFacilitator*: This method is used to inform the Coordinator that a registered Facilitator is going offline, i.e., its hosting device is leaving the MPMS.

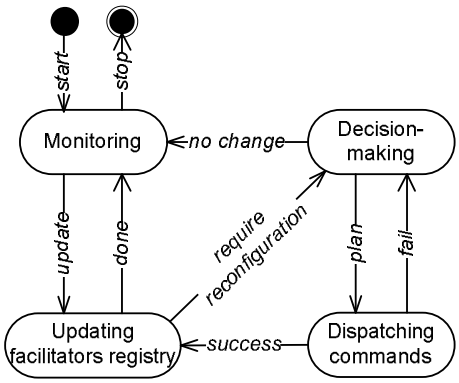

Figure 6.6: *The state diagram of Coordinator*

To deal with the potential risk that the connection between a registered Facilitator and the Coordinator unexpectedly becomes unavailable, a proper time-out mechanism is included. Using this time-out mechanism, the Coordinator can discover an unreachable Facilitator and remove it from the Facilitator registry.

- *bspuBlocked*: This method is used to inform the Coordinator that a BSPU is successfully blocked.

- *bspuQuiescent*: This method is used to inform the Coordinator that a BSPU is now quiescent, i.e., a special "active" state.

<<interface>> ICoordinatorRegistration
+ registerFacilitator(fURL, fContext): fID + updateFacilitator(fID, fContext) + unregisterFacilitator(fID) + bspuBlocked(bspuID,fID) + bspuQuiescent(bspuID,fID)

Figure 6.7: *The interface of Coordinator Registration. "fURL" is the connection address of a Facilitator. "fContext" describes the current context information of a Facilitator's hosting device. "fID" is a Coordinator-wide unique identifier for a Facilitator*

6.2 Affected Tasks

Besides calculating the optimal task assignments, the other main responsibility of the Coordinator is to automatically generate a reconfiguration plan given a current task assignment ("α") representing the current MPMS system configuration, and a target task assignment ("β") representing the identified new system configuration. An example of these two different task assignments for the system modeled in Figure 6.8 are illustrated in Table 6.1.

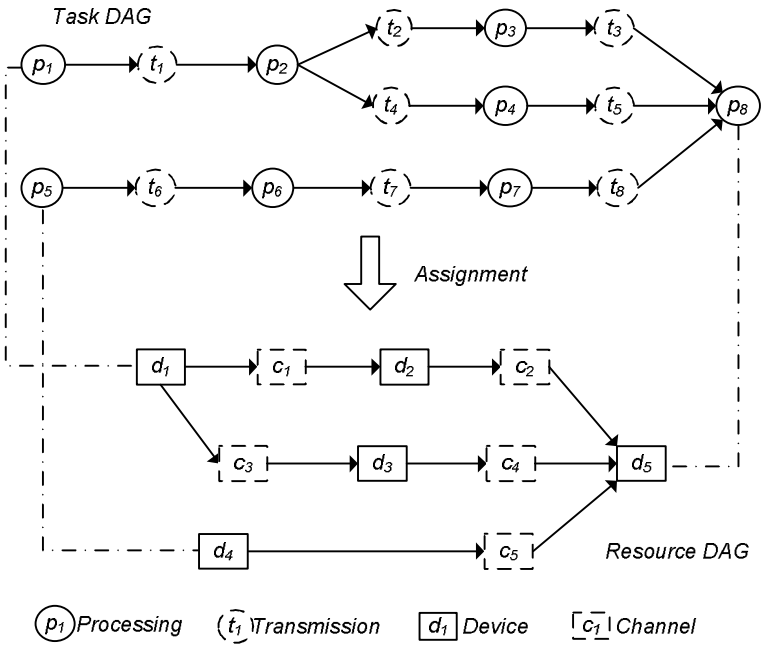

Figure 6.8: *Copy of Figure 3.4*

An original task assignment and a target task assignment are different if at least one task's location is changed, e.g., addition, relocation or removal. In the example illustrated in Table 6.1, the locations of task "p_2", "t_1", "t_2" and "t_4" are changed. Additions or removals occur when new tasks are introduced or existing tasks are removed. In a set of connected tasks, a location change of one task will require adjustments to other tasks that connect to this task. For example, as shown in Figure 6.8, if the location of task "t_2" or task "t_4" is changed, some adjustment should be performed at "p_2" in order to guarantee that the output of "p_2" is correctly connected to "t_2" or "t_4". Thus, to generate a reconfiguration plan, we should not only consider the tasks whose locations are explicitly changed, but also those tasks that

	α(original)	β(target)
p_1	d_1	d_1
p_2	d_1	d_5
p_3	d_5	d_5
p_4	d_5	d_5
p_5	d_4	d_4
p_6	d_4	d_4
p_7	d_4	d_4
p_8	d_5	d_5
t_1	d_1	d_1, c_1, d_2, c_2, d_5
t_2	d_1, c_1, d_2, c_2, d_5	d_5
t_3	d_5	d_5
t_4	d_1, c_1, d_2, c_2, d_5	d_5
t_5	d_5	d_5
t_6	d_4	d_4
t_7	d_4	d_4
t_8	d_4, c_5, d_5	d_4, c_5, d_5

Table 6.1: *Two different task assignments "α" and "β" for the system shown in Figure 3.4. The affected tasks are highlighted.*

are affected implicitly. In the reconfiguration from "α" to "β" (Table 6.1), not only "p_2", "t_1", "t_2" and "t_4" should be taken care of, but also the adjustment to "p_1" should be included in the reconfiguration plan. 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 removed, or (2) any of its direct successor transmission tasks is affected.

Thus we can identify, for any reconfiguration, the set of affected tasks unambiguously as shown in Table 6.1. In the task distribution infrastructure, a reconfiguration plan only needs to address those processing BSPUs that implement affected processing tasks and those relaying BSPUs that implement affected transmission tasks. We refer to these processing BSPUs and relaying BSPUs as affected BSPUs.

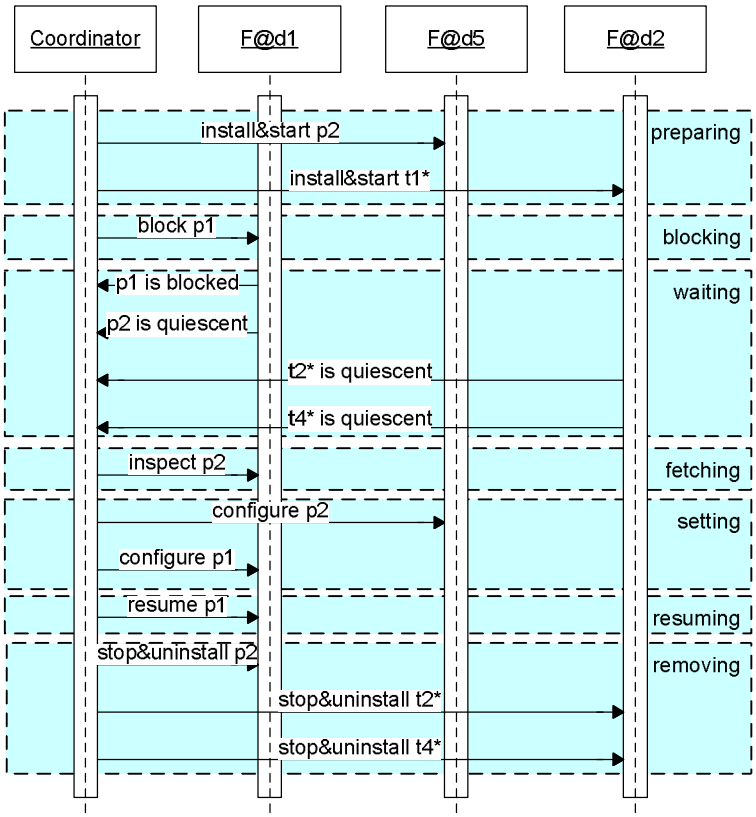

Figure 6.9: *The seven-step reconfiguration plan applied to support the reconfiguration from "α" to "β"*

6.3 Reconfiguration Plan

In order to maintain state consistency during a dynamic reconfiguration, a three-stage reconfiguration approach has to be followed [AVSN01]: (1) drive system to a safe state in which to-be-reconfigured components are self-contained and stable and none of them is involved in any interaction; (2) detect if a 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 atomic transactions: in case 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 reconfiguring from "α" to "β" as depicted in Figure 6.9:

1. Preparing: Before interrupting the on-going telemonitoring application, the Coordinator installs and starts every relocated and newly added BSPU at the targeted new location, e.g., "install&start p_2" and 'install&start t_1*" in Figure 6.9. Thus, in case any action fails in this step, there is no penalty on the on-going telemonitoring application.

2. Blocking: In this step, the on-going telemonitoring application is interrupted. the Coordinator identifies and blocks all the so-called *source affected BSPUs* in the original configuration, e.g., "block p_1". A *source affected BSPU* is an affected BSPU none of whose predecessor BSPUs are affected.

3. Waiting: after a source affected BSPU enters the "Blocked" state, it sends a notification via its hosting Facilitator to the Coordinator, e.g., "p_1 is blocked". After a BSPU receives a "Blocked" flag from its predecessor BSPU and finishes all possible processing of the remaining data stream in its input buffer, it enters the "Quiescent" state. This BSPU notifies the Coordinator via its hosting Facilitator about this situation, e.g., "p_2 is quiescent". After all the notifications from affected BSPUs are received by the Coordinator, the system enters a safe state for the next step.

4. Fetching: The Coordinator inspects the current execution state information from relocated BSPUs in the original configuration, e.g., "inspect p_2". Now, the Coordinator has the execution state information of relocated BSPUs.

5. Setting: The Coordinator sets the execution state of every relocated BSPU at the targeted new location, e.g., "configure p_2". The Coordinator sets the BSPU whose configuration parameters are required to update in order to maintain the connection correctly, e.g., "configure p_1".

6. Resuming: The Coordinator resumes all source affected BSPUs, e.g., "resume p_1". After this step, the MPMS resumes to work normally in the target configuration.

7. Removing: The Coordinator removes all the outdated BSPUs (including relaying BSPUs) left from the original configuration, e.g., "stop&uninstall p_2" and "stop&uninstall t_2* and t_4*".

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.

6.4 Reconfiguration Cost

In this section, we present a computational model to evaluate the reconfiguration cost in terms of reconfiguration time, which is the elapsed time between the moment that Coordinator initiates a new reconfiguration and the moment that the reconfiguration finishes and the system starts running under a new configuration. The purpose of this model is twofold. (1) The stateful reconfiguration could cause a potential disruption at the telemonitoring application. It is 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 multiple new task assignments that offer similar performance compared to the optimal one under the new situation (c.f. Section 3.3.3). To choose the best, we need to balance the tradeoff between the performance gain of these new task assignments and their reconfiguration complexity.

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 reconfiguration communications and actions. We define the following cost functions to model 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 set of communication channels between the Coordinator and Facilitators. From the Coordinator point of view, we label "communication channel between Coordinator and Facilitator" simply as "Facilitator". In our model, we denote Facilitators according to its location. For example, the Facilitator located at device "d_5" is denoted as "$F@d_5$".

- $c^a(Facilitator, BSPU, Action)$ 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

  $Action = \{block, resume, install, uninstall, start, stop, inspect, configure, wait\}$

The Coordinator's knowledge about these two cost functions can be learnt over time through monitoring 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 estimates. In Table 6.2,

Step	Cost
preparing	$c^x(F@d_5) + c^a(F@d_5, p_2, install) + c^a(F@d5, p_2, start)$ $+c^x(F@d_2) + c^a(F@d_2, t_1*, install) + c^a(F@d2, t_1*, start)$
blocking	$c^x(F@d_1) + c^a(F@d_1, p_1, block)$
waiting	$c^x(F@d_1) + c^a(F@d_1, p_1, wait) + c^x(F@d_1) + c^a(F@d_1, p_2, wait)$ $+c^x(F@d_2) + c^a(F@d_2, t_2*, wait) + c^x(F@d_2) + c^a(F@d_2, t_4*, wait)$
fetching	$c^x(F@d_1) + c^a(F@d_1, p_2, inspect)$
setting	$c^x(F@d_5) + c^a(F@d_5, p_2, configure) + c^x(F@d_1)$ $+c^a(F@d_1, p_1, configure)$
resuming	$c^x(F@d_1) + c^a(F@d_1, p_1, resume)$
removing	$c^x(F@d_1) + c^a(F@d_1, p_2, stop) + c^a(F@d_1, p_2, uninstall)$ $+c^x(F@d_2) + c^a(F@d_2, t_2*, stop) + c^a(F@d_2, t_2*, uninstall)$ $+c^x(F@d_2) + c^a(F@d_2, t_4*, stop) + c^a(F@d_2, t_4*, uninstall)$

Table 6.2: *Stepwise reconfiguration costs from from "α" to "β"*

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 6.9. In the proposed reconfiguration plan, only from step 2 (blocking) to step 6 (resuming), the telemonitoring application potentially suffers from a service disruption. Thus, the total reconfiguration cost of a particular reconfiguration in our case is the combination of the costs of these 5 steps.

6.5 Experiment Results

We implemented the task distribution infrastructure based on the OSGi technology. A 3rd party OSGi added-on service, called R-OSGi[2], is used in our implementation to provide the remote connection support between distributed OSGi frameworks based on the Service Location Protocol (SLP). There are several existing open source OSGi framework implementations. We selected "Equinox"[3] which is the first certified implementation by the OSGi consortium and it is also the cornerstone of the "Eclipse" development tool. BSPUs, Facilitator, and Coordinator are all implemented as OSGi bundles (services). Two different device configurations, i.e., Facilitator device and Coordinator device, are illustrated in Figure 6.10. Both device configurations require the presence of a bundle named "FacilitatorContro-

[2]http://r-osgi.sourceforge.net/
[3]http://www.eclipse.org/equinox/

lAPI". This bundle declares the Facilitator Control interface (c.f. Figure 6.4) so that the Coordinator service can bind and invocate a Facilitator service on a remote OSGi framework. At the Facilitator device, both the invocations to the Coordinator Registration Interface (c.f. Figure 6.7) and the time-out mechanism are supported by the "EventService" bundle:

- When a Facilitator wants to register, update or unregister itself, it will send an event message carrying the relevant information, e.g., the address and context information.
- When a BSPU wants to inform the Coordinator that it entered a "Blocked" or "Quiescent" state, it will send a 'Blocked" or "Quiescent" event message.
- Periodically, a Facilitator also sends out a keep-alive event message to inform the Coordinator that it is still available. This is used to assist the Coordinator correctly maintaining a list of available Facilitators in an MPMS.

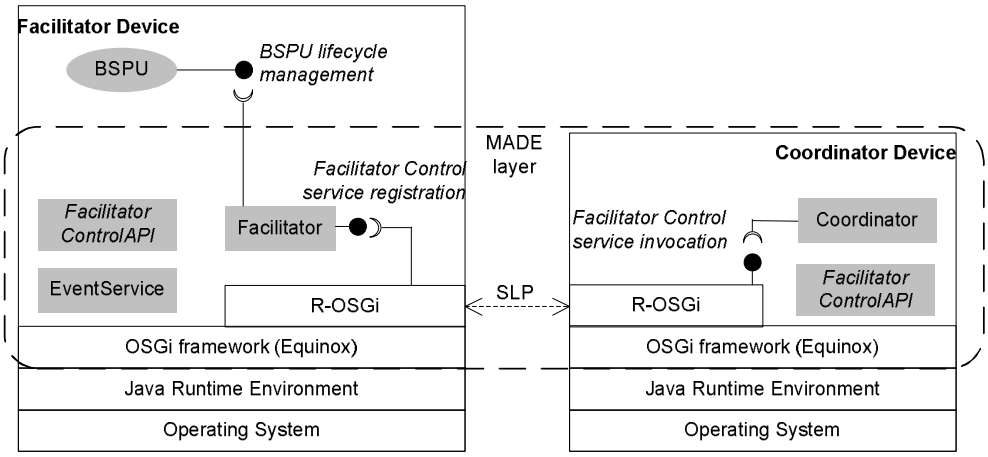

Figure 6.10: *Coordinator device connecting Facilitator device through R-OSGi service interface*

To validate the sequential seven-step dynamic reconfiguration scheme, in particular to evaluate the reconfiguration cost, we measured the dynamic reconfiguration (Figure 6.9) from task assignment "α" to "β". Our experimental setting consists one laptop and three PCs where three PCs are located in the same LAN and the laptop is connected through Wireless LAN (802.11g).

1. The Coordinator device is a laptop with an Intel Duo Core 2.5GHz CPU, 4GB memory, and running Windows Vista.

Step	Mean time (ms)	std time (ms)
preparing	57.41	19.55
blocking	9.23	12.44
waiting	233.86	78.82
inspecting	31.86	14.75
setting	64.64	13.10
resuming	54.50	18.59
removing	25.68	14.12

Table 6.3: *Measured time duration of each reconfiguration step in the experiment of reconfiguring from the assignment "α" to the assignment "β" as shown in Figure 6.9*

2. The Facilitator device "d_1" is a PC with an Intel Duo Core 2.66GHz CPU, 1GB memory and running Windows XP 2002 SP2.

3. The Facilitator device "d_2" is a PC with an Intel Duo Core 2.4GHz CPU, 2GB memory and running Windows XP 2002 SP2.

4. The Facilitator device "d_5" is a PC with an Intel Duo Core 2.4GHz CPU, 1GB memory and running Windows XP 2002 SP2.

We ran this dynamic reconfiguration experiment 20 times and report the average time duration and its standard deviation for each reconfiguration step in Table 6.3. We observed the main bottleneck for dynamic reconfiguration is on the "waiting" step. Even for the powerful computers, broadband connection, and the relatively simple reconfiguration plan we used in the experiment, the average time duration for the "waiting" step is still 234 milliseconds. The average total service disruption caused by the steps from "blocking" till "resuming" is 394 milliseconds. We can expect the level of service disruption for a real MPMS to be even higher, e.g., close to the level of disruption observed in the MoiPADS system (c.f. Section 2.1.1). Without the detailed analysis on telemonitoring application and its requirement, it is not easy to make the conclusion whether this kind of service disruption will affect the provided healthcare service or not. Using appropriate recording and buffering technologies, we could guarantee that the transmitted and processed bio-signal will not be lost. However, a certain level delay caused by the reconfiguration's service disruption is, very likely, not avoidable. Hence, depending on the specific data delivery requirement, some MPMSs may just tolerate the excessive delay while other MPMSs may require a warning to be given in order to let the user take proper actions.

6.6 Concluding Remarks

This chapter details the design of the MADE middleware and presents a prototype implementation based on the OSGi technology. There are three kinds of architectural components in MADE, i.e., Coordinator, Facilitator and BSPU. In an MPMS, a Coordinator and a set of Facilitators form an overlay network on top of the m-health platform and work together in a cooperative fashion to achieve task distribution based adaptation. To support the distributed execution of telemonitoring applications, we introduced BSPU that plays the role of a unit of system composition. A BSPU receives continuous bio-signal streams via its input buffer from its predecessor BSPUs, manipulates the streams, and transfers the processed output streams to the successor BSPUs if any. The architectural and behavioral design of these three components are presented in Section 6.1.

Another main contribution of this chapter is that we defined a scheme to construct reconfiguration plans for MPMSs. First, in order to identify the difference between the current task assignment and a target task assignment, Section 6.2 defines the concept of affected task and explains how affected tasks can be identified. Determining the affected tasks is the essential step for making the reconfiguration plan since they form the targeted area of a system reconfiguration. Section 6.3 introduces a 7-step scheme to construct reconfiguration plans. This scheme first drives the MPMS to a safe state in which affected BSPUs are self-contained and none of them is involved in any interaction. Only after a safe state is detected (all affected BSPUs are either in a "Blocked" state or a "Quiescent" state), the reconfiguration towards a target task assignment can start.

As discussed earlier on dynamic reconfiguration (Section 2.3), a stateful mechanism that supports execution state transfer will, unavoidably, have certain impact on the system and its provided services. Section 6.4 presents a concept of reconfiguration cost (in terms of reconfiguration time) to deal with this service disruption problem. Section 6.5 presents the experiment results obtained from our prototype implementation. We observed a hundred-millisecond level service disruption in our experimental setting and expect the disruption to be larger in a real-world MPMS. There are different approaches to tackle this unavoidable service disruption, e.g., buffering transmitted data or alerting the users. Furthermore, since the main bottleneck in the dynamic reconfiguration is on the "waiting" step, developers could pay more attention on improving the specific designs related to this step when dealing with a concrete system. Depending on the specific telemonitoring applications and their medical requirements, these design choices may vary and they are not the focus of this thesis.

Chapter 7

Conclusion

This chapter presents the conclusion of this thesis. Section 7.1 presents general considerations and Section 7.2 explains our findings with respect to the proposed research questions in Chapter 1. Section 7.3 summarizes the most important research contributions of this thesis. Section 7.4 discusses several topics that are of interest for future research.

7.1 General Considerations

We recognize that, similar to other applications operating in a mobile environment, the performance of MPMS could be (deeply) affected by context changes and scarcity of resources, e.g., network bandwidth, battery power, and computational power of handhelds. When a mismatch between the application's demand and the platform's resource supply exceeds a certain threshold, an entire MPMS may fail in responding accurately and timely to an emergency. Thus, the success of an MPMS relies heavily on whether the system can provide adequate and continuous bio-signal processing and transmission services despite context changes. Finding a suitable solution to address the demand and supply mismatch problem in MPMSs was the key motivation for this PhD research project.

We proposed an adaptation mechanism to adjust the assignment of tasks across available devices at run-time. This design was referred to in this thesis as task-distribution-based adaptation mechanism. The rationale is that, at a particular point in time, if one device cannot support a task for its computation or data communication demands, some other devices with richer resources might take over. The advantage over traditional methods is that the user requirements are less likely to be compromised and distributed resources can be better utilized. Following this approach, this thesis covers two major research topics: (1) computation of a suitable task assignment, and (2) dynamic distribution of tasks across devices according to this new assignment at run-time.

7.2 Research Questions Revisited

In Section 1.3, we presented a list of research questions that are of importance for this PhD thesis and these questions have been addressed in various chapters. In this section, we collect and summarize our answers.

RQ1: What do we require for "a suitable task assignment" in an MPMS?

Given an MPMS, i.e., an m-health platform and deployed telemonitoring applications, there are a number of possibilities to configure the system based on different task assignments. For a particular application and its users, "a suitable task assignment" implies different system requirements reflecting users' specific needs. In Section 3.1, we proposed requirements at four levels ranging from elementary to advanced. They are:

- Level 1: A suitable task assignment should be able to operate in the present situation.
- Level 2: Including the level 1 requirement, a suitable task assignment should be able to operate in a foreseeable future with a sufficiently high probability.
- Level 3: Including the level 2 requirement, a suitable task assignment should satisfy all performance assurance requirements.
- Level 4: Including the level 3 requirement, a suitable task assignment should satisfy the performance optimization requirement on a particular measure.

We leave the final choice to the developers of an MPMS to decide which level of task assignment they like to associate with their designs. In this thesis, we provided task assignment solutions with the highest requirement, i.e., level 4. Furthermore, as we study the reconfiguration cost, it becomes clear that even the optimal task assignment computed based on the new context information might not be the real suitable one by taking into consideration the reconfiguration costs. To tackle this problem, we formulated the task assignment problem as to obtain several candidate task assignments instead of only searching for the optimal assignment. In this thesis, we admit that it is not always possible to find a suitable task assignment given the particular context and requirements. When a suitable task assignment does not exist, other means have to be used to ensure that users are aware of the requirement violations.

RQ2: What benefits can we expect from adapting an MPMS?

Closely coupled with the four-level requirements of suitable task assignments, we summarize the expected benefits of task-distribution-based adaptation mechanism in the following:

- If an adaptation is performed in the event of an instant mismatch, e.g., required data throughput exceeds the available bandwidth at a certain channel, a new task assignment should be identified to eliminate this mismatch. Thus, the first performance gain is that the system under the new configuration can still operate while the old configuration will fail immediately.
- Although the technique of forecasting a potential mismatch is not the focus of this thesis, recent work has shown the possibility of highly accurate user behavior prediction. For example, in [ECQ09], the authors presented a Bayesian network based method that can predict a person's movement with over 90% accuracy. Hence, we argue that identifying a potential mismatch with a high confidence level is becoming a possibility. An adaptation to avoid it will help the system to deliver its promised service in the direct foreseeable future.
- Due to a context change, it may happen that the current task assignment no longer can meet the performance assurance requirement. Thus, adapting the system configuration towards one of the task assignments that can satisfy these constraints will help an MPMS to fulfill the performance assurance requirement.
- As we demonstrated in Chapter 5, performance improvements can be expected if we allow dynamic reconfiguration according to an optimal task assignment. For example, for the battery lifetime, a dynamic approach can increase the system battery lifetime more than 200% compared to the static setting.

RQ3: How to design effective yet efficient task assignment algorithms for MPMSs?

Similar to earlier studies [NT93, KA98], the task assignment problem in MPMSs is NP-hard. However, in case the task model and resource model have a certain topology and the cost model satisfies certain properties, dynamic programming methods can be applied and polynomial-time algorithms exist. In this thesis, we proposed two graph-based polynomial-time algorithms, which deal with the "chain-chain assignment" and the "tree-star assignment" respectively. When the targeted MPMS complies with such specific cases, we recommend the use of these graph-based polynomial-time algorithms since they provide an optimal solution, yet the

computation time is bounded. Although the proposed algorithm for "chain-chain assignment" is efficient with regard to the computation time, it still suffers a major drawback on memory consumption. We proposed a GA-based solution that is more friendly on memory usage as an alternative. For the more general "DAG-DAG assignment", we proposed the A*-based task assignment algorithms in Chapter 5.

RQ4: How to support dynamic task distribution in MPMSs?

In dynamic reconfiguration of an MPMS, 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, the proposed task distribution infrastructure should support execution states transfer, i.e., stateful reconfiguration. To achieve this, the following designs and implementations are presented in this thesis:

- The OSGi technology is selected as the implementation framework. A software component, named BSPU, is proposed for implementing the distributable tasks. In the proposed lifecycle of a BSPU, we introduced two new states and two new transitions compared to the standard OSGi bundle lifecycle. These newly added controls can be used to temporarily freeze a set of BSPUs, force them into a consistent state, and allow a stateful reconfiguration.

- Following a standard approach to maintain state consistency during a dynamic reconfiguration, we proposed a sequential seven-step scheme to create dynamic reconfiguration plan.

RQ5: What is the potential disruption caused by dynamic task distribution and how to measure this reconfiguration cost?

Dynamic system reconfiguration, especially when the transfer of execution state is required, often has a negative impact on the on-going services provided by the system. It is important to understand the potential service disruption caused by reconfiguration and to estimate reconfiguration cost. This thesis presents a model to evaluate the reconfiguration cost in terms of reconfiguration time, which is the elapsed time between the moment a new target task assignment is found and a reconfiguration is triggered and the moment that the reconfiguration finishes and the system starts running under a new configuration. We treated every reconfiguration plan as a set of control communications and control actions. Thus, the entire reconfiguration cost can be viewed as the sum of the cost of each individual control action and control communication. In Section 6.4, we have considered two cost functions

to capture this information and the knowledge about these two cost functions can be learnt over time through monitoring system operation.

7.3 Research Contributions

The main contributions of this thesis are:

- A graph-based system model and a set of task assignment algorithms using a variety of techniques; and
- A middleware layer solution to support task distribution, which includes interface and protocol design, a scheme to construct reconfiguration plans, and a model for estimating reconfiguration cost.

These contributions are discussed in the sequel.

7.3.1 Task Assignment Algorithms

This section summarizes the research work on task assignment algorithms in this thesis.

Graph Based Model

Similar to earlier research on the task assignment problem, we use a graph-based modeling approach in this thesis (Section 3.3.3). An m-health platform is modeled as a resource DAG, in which we distinguish two types of resources: device vertices and channel vertices. We model a telemonitoring application as a task DAG and we distinguish two types of streaming tasks: (1) stream processing tasks that typically perform some operation on the bio-signal stream, and (2) stream transmission tasks that are the glue between processing tasks. The benefit of modeling transmission tasks explicitly are two-fold. First, it allows us to easily characterize properties of the data stream. Second, transmission tasks can be mapped onto a directed communication path in the resource DAG, hence permitting the existence of relaying devices.

Chain-Chain

Section 4.1 presents an efficient task assignment algorithm when both the telemonitoring application and the m-health platform form a chain. Compared to earlier

work, this new method relaxes a so-called contiguity constraint, which is a necessity in the earlier case but unnecessarily constraining MPMSs. Furthermore, we observed from our experiments that the assignment graph "construction" step is more time consuming than the actual "search" step in most of cases, which suggests that the evaluation on this kind of graph-based method should be based on the time complexity of the entire method instead of the "search" step only [Bok88, SC90, NO91].

Although this graph-based exact algorithm is more efficient compared to earlier work, it still has a major drawback on memory consumption: it requires an assignment graph to be constructed first, which is a bottleneck of the solution (Table 4.1). In Section 4.2, we also investigated GA based solutions. Although, GA-based algorithms can not guarantee an optimal solution, they have much smaller and bounded requirements on memory space while providing satisfactory results.

Tree-Star

As motivated by Bokhari [Bok88], many distributed application settings can be modeled as a mapping of a task tree onto a star network (Section 2.2.3). In MPMS, this tree-star model is popular as well [MPW07]. However, it is not possible to directly apply the existing algorithms to our problem due to the assumptions and constraints associated with them. Section 4.3 presents a new tree-star assignment algorithm. Our work differs from Bokhari's original approach and some follow-up studies in the following aspects:

1. The following three constraints (defined earlier by Bokhari) are relaxed in our algorithm by a coloring scheme:
   - There are as many satellite devices as there are leaf vertices in the task tree and it is possible not to use them if the optimal assignment dictates so.
   - All the satellite devices are homogeneous.
   - If two tasks are assigned to a satellite device, their lowest common ancestor is also assigned to the same satellite device.
2. Bokhari proposed the SB algorithm to find a partition that minimizes the bottleneck processing time while our goal is to find a partition that minimizes the end-to-end processing delay.

DAG-DAG

In Chapter 5, we proposed A*-based task assignment algorithms. The algorithm performance is evaluated based on Java implementations. In our experiments, we

observed a significant performance improvement if the system can be dynamically reconfigured according to an optimal task assignment. For example, the system battery lifetime resulted from a dynamic approach, sometimes, is more than tripled compared to a static approach. Furthermore, there is a clear relation between the improved performance and the context change, i.e., the more the context changes, the better improvement can be expected. When dealing with a larger problem, the bounded version of the A*-based algorithm is shown to be an effective (in terms of the quality of the solution) and efficient (in terms of the required computational resource) alternative solution.

7.3.2 Task Distribution Infrastructure

To support dynamic task distribution and to hide the heterogeneous nature of MPMSs, a middleware level solution is proposed.

MADE

As presented in Section 3.2, the task distribution middleware (MADE) provides four main functionalities: *Monitoring*, *Analysis*, *Decision*, and *Enforcement*. The Monitoring functionality is responsible for the telemonitoring application registration, device discovery, resource monitoring, and context discovery/registration. The Analysis functionality takes the observed knowledge about telemonitoring application and m-health platform as input and runs a task assignment algorithm to determine the assignment with optimal system performance. The Decision functionality compares the computed optimal assignment with the current system configuration to determine the actual cost of reconfiguration. If the reconfiguration cost can be covered by the enhanced performance of the new configuration, the new assignment plan will be executed. The Enforcement functionality controls the MPMS to adjust its configuration according to the new assignment. Please note that the high-level design of MADE is for experimental purposes, i.e., to validate the reconfiguration approach and the task assignment algorithms, and thus is not fully optimized.

Reconfiguration Plan

An original task assignment and a target task assignment are different if at least one task location is changed (c.f. Table 6.1). In this thesis, we argue that we should not only consider the tasks whose locations are explicitly changed, but also those tasks that are affected implicitly: configuration updates of these implicitly affected tasks should also be included in a reconfiguration plan. Inspired by earlier discussions on dynamic reconfiguration (c.f. Section 2.3), a sequential seven-step scheme to create

reconfiguration plan is proposed in Section 6.3. This scheme can be categorized as a stateful mechanism since it supports preserving the system correctness without the knowledge of detailed application information. We validated this design by means of an OSGi-based prototype implementation. The execution results proved that our proposed task distribution infrastructure and reconfiguration plan are indeed correct and effective. Although a certain level delay is unavoidable in telemonitoring services, we argue that, by using appropriate recording and buffering technologies, the transmitted and processed bio-signal will not be lost. Hence, depending on the specific data delivery requirement, it is possible to design and build MPMSs that can tolerate this excessive delay.

Reconfiguration Cost

Every 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. In this thesis, we defined two cost functions to model the time duration of each control communication and control action observed at the Coordinator. The values of these two cost functions can be learnt over time through monitoring system operation. Hence, it is feasible now for the Coordinator to make the decision on what is the best task assignment by taking into consideration both the estimated performance improvement and the associated reconfiguration cost.

7.4 Future Research

This section lists several issues for future research.

Task Profiling in Telemonitoring

In this thesis, we presented graph-based models for telemonitoring applications and m-health platforms. We did not discuss on how to obtain the values of these labels and we admit that it is not so trivial to do so, especially in a mobile and heterogeneous environment like MPMS. It is not easy to derive a task's profiling information without precise knowledge of its targeted device. The diversity of processor design, Instruction Set Architecture (ISA), and compilation tools complicate this problem further [AS96, HP03]. This difficulty has not been emphasized sufficiently in earlier research on task assignment and we see a considerable effort should be made here. Earlier work on analytical benchmarking and task profiling [MS98] present possible approaches. For example, following a task profiling approach, we could conduct a

trial-based project. In this project, each task in MPMS is tested against every possible type of device and network channel. Then the operation history is recorded and the derived knowledge of its resource demand is put into the task's profile.

Extension on Chain-Chain Task Assignment Algorithms

Our general problem formulation shows that the goal of task assignment algorithms is to find a set of candidate assignments that provide the best performance. This requirement is reflected by the proposed A*-based algorithms in Chapter 5. For the problems with simpler topology, e.g., "chain-chain assignment", the studied algorithms do not provide multiple candidate assignments and only address the performance measure of end-to-end delay. Thus, there are two possible directions to extend chain-chain task assignment algorithm.

The first extension is to provide multiple candidate assignments. Based on the currently proposed chain-chain assignment algorithm on minimizing end-to-end delay (Section 4.1), a straightforward solution is to apply a "K-shortest path algorithm" [Epp98]. Instead of finding the shortest path in the constructed task assignment graph that represents the optimal assignment, we could search for k-shortest paths ($k > 1$) representing k candidate assignments that provide best performances among all possible task assignments.

The second extension is to study algorithms addressing other performance measures, e.g., system battery lifetime. Actually, we did propose a simple algorithm on chain-chain assignment problem with the objective of maximizing system battery lifetime in a separate paper [JMB+07]. However, the model in that paper is very limited in the sense that the contiguity constraint can not be relaxed. We have not yet found a polynomial-time exact algorithm that relaxes this contiguity constraint. It seems that due to the proposed method of estimating system battery lifetime (Section 3.3.2), the dynamic programming approach is not applicable anymore. Thus, it is unlikely that a polynomial-time exact algorithm will ever be found. On the other hand, GA-based approaches can be applied easily to design an algorithm addressing other performance measures. But its effectiveness and efficiency are yet to be proven.

New Heuristic Functions in The A*-based Algorithms

We do not claim that the function $h(n)$ defined in Chapter 5 is the exact lower-bound of the additional cost. In fact, any estimation that can be proven to be smaller than the exact lower-bound can work. The extreme example here is to define $h(n)$ as "0". Then, this A*-algorithm with "$h(n) = 0$" becomes a pure best-first algorithm. Since we did not strive for the exact lower-bound function, we acknowledge the

possibility of finding new heuristic functions that are closer to the real lower-bound. In such cases, a performance improvement of the proposed A*-based algorithms may be achieved.

Another possible extension is to apply a so-called "bidirectional search" mechanism [RN02]. To put it in a simple way, "bidirectional search" runs two simultaneous searches. The first one moves forward from the initial state, e.g., assigning a source task. The second one moves backward from the goal state, e.g., assigning a sink task. The entire search stops when the two separate search procedures meet in the middle. In theory, despite the increased design complexity, this bidirectional approach could reduce the search time.

Incorporate Reconfiguration Cost Into Task Assignment Algorithms

As we proposed in this thesis, task assignment algorithms and reconfiguration cost are treated separately. If a system reconfiguration is required, we first use a selected task assignment algorithm to obtain a set of candidate assignments. Then we construct the reconfiguration plan based on the 7-step scheme for each candidate task assignment. Using the proposed computational model we could estimate the reconfiguration cost of every plan. Last, by comparing each plan's cost with its performance gain, we can determine which task assignment should be the target assignment and start executing its associated reconfiguration plan.

If we could integrate these two functional aspects together into a new design, i.e., a new task assignment algorithm that can incorporate the comparison with reconfiguration cost, we can expect a significant improvement. Take the A*-based algorithm as an example, every path from the root node to a leaf node in the search tree corresponds to a task assignment. If the algorithm could somehow "remember" the path representing the current task assignment before constructing and traversing the search tree, it should be possible for the algorithm to compare the difference between each new found task assignment with the current task assignment while the search is being performed. Thus, with just one step of executing a task assignment algorithm, potentially we could obtain the real suitable task assignment by taking into consideration reconfiguration cost.

Refine The Task Distribution Infrastructure

In Chapter 6, we presented the task distribution infrastructure of the MADE middleware that supports the dynamic reconfiguration and a sequential seven-step scheme to create a reconfiguration plan. We noted that the high-level design of this task distribution infrastructure was for experimental purposes and has not been fully

optimized. A future direction is therefore to seek for a refined design and more optimized coordination processes. By doing so, we could reduce the overhead cost of running the MADE middleware in an MPMS and leave stringent resources in an MPMS to its main functions. In particular, additional future efforts can be made to optimize the control communications and control actions performed in the "waiting" step.

Distributed MPMSs With A Large Patient Group

In this thesis, we studied the task-distribution-based adaptation mechanism based on a single patient scenario. It is unavoidably that, when a large patient group starts using telemonitoring services, we will have to support distributed MPMSs [PvBMH09]. This is much more complicated problem since some m-health platform devices, e.g., back-end server, will be shared by multiple patients and the applied adaptation mechanism should take this into account. A possible direction based on the current Coordinator/Facilitator design is to introduce a higher-level Coordinator that can control the distributed individual patient specific Coordinator. Such a higher-level Coordinator can have a global view of the entire distributed MPMSs and assist each MPMS to adapt in a coherent and consist manner.

Bibliography

[AAH05] W. Alsalih, S. Akl, and H. Hassanein. Energy-aware task scheduling: Towards enabling mobile computing over MANETs. *19th International Parallel and Distributed Processing Symposium (IPDPS'05)*, 2005.

[AJS+06] L. Amini, N. Jain, A. Sehgal, J. Silber, and O. Verscheure. Adaptive control of extreme-scale stream processing systems. In *26th IEEE International Conference on Distributed Computing Systems (ICDCS'06)*, page 71, 2006.

[All04] The Telemedicine Alliance. Telemedicine 2010: Visions for a personal medical network. Technical report, 2004.

[AS96] M. Atkins and R. Subramaniam. PC software performance tuning. *IEEE Computer*, 29(8):47–54, 1996.

[AVSN01] J. P. Almeida, M. Van Sinderen, and L. Nieuwenhuis. Transparent dynamic reconfiguration for CORBA. In *The Third International Symposium on Distributed Objects and Applications (DOA'01)*, page 197, Washington, DC, USA, 2001. IEEE Computer Society.

[BB04] B.J. Bonfils and P. Bonnet. Adaptive and decentralized operator placement for in-network query processing. *Telecommunication Systems*, 26(2-4), June 2004.

[BFK+00] B. Badrinath, A. Fox, L. Kleinrock, G. Popek, P. Reiher, and M. Satyanarayanan. A conceptual framework for network and client adaptation. *Mobile Networks and Applications (MONET)*, 2000.

[Bok79] S. H. Bokhari. Dual processor scheduling with dynamic reassignment. *IEEE Transactions on Software Engineering*, SE-5(4):341–349, 1979.

[Bok81] S. H. Bokhari. On the mapping probelm. *IEEE Transactions on Computers*, 30(3), 1981.

[Bok88] S. H. Bokhari. Partitioning problems in parallel, pipelined, and distributed computing. *IEEE Transactions on Computers*, 37(1):48–57, 1988.

[BR08] A. Benoit and Y Robert. Mapping pipeline skeletons onto heterogeneous platforms. *Journal of Parallel and Distributed Computing*, 68(6):790–808, 2008.

[BSBB98] T. D. Braun, H. J. Siegel, N. Beck, and L. Blni. A taxonomy for describing matching and scheduling heuristics for mixed-machine heterogeneous computing systems. In *17th IEEE Symposium on Reliable Distributed Systems*, 1998.

[CC03] Alvin T.S. Chan and S. N. Chuang. MobiPADS: A reflective middleware for context-aware mobile computing. *IEEE Transactions on Software Engineering*, 29(12):1072–1085, 2003.

[CDK+04] K. Cooper, A. Dasgupta, K. Kennedy, C. Koelbel, A. Mandal, G. Marin, M. Mazina, J. Mellor-Crummey, F. Berman, H. Casanova, A. Chien, H. Dail, X. Liu, A. Olugbile, O. Sievert, H. Xia, L. Johnsson, B. Liu, M. Patel, D. Reed, W. Deng, C. Mendes, Z. Shi, A. YarKhan, and J. Dongarra. New grid scheduling and rescheduling methods in the GrADS project. In *18th International Parallel and Distributed Processing Symposium (IPDPS'04)*, page 199, 2004.

[CFR99] R.C. Correa, A. Ferreira, and P. Rebreyend. Scheduling multiprocessor tasks with genetic algorithms. *IEEE Transactions on Parallel and Distributed Systems*, 10(8):825–837, Aug 1999.

[Chr75] N. Christofides. *Graph Theory: An Algorithmic Approach*. Academic Press Inc, 1975.

[CK88] T. L. Casavant and J. G. Kuhl. A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Transactions on Software Engineering*, 14(2):141–154, 1988.

[CLRS01] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms (2nd Edition)*. MIT Press and McGraw-Hill, 2001.

[CNNS94] A. N. Choudhary, B. Narahari, D. M. Nicol, and R. Simha. Optimal processor assignment for a class of pipelined computations. *IEEE Transactions on Parallel and Distributed Systems*, 5(4):439, 1994. 1045-9219.

[Cos07] P. D. Costa. *Architectural support for context-aware applications: from context models to services platforms*. PhD thesis, Univ. of Twente, 2007.

[Cou00] G. Coulson. What is reflective middleware? *IEEE Distributed Systems Online*, 2000.

[DP85] R. Dechter and J. Pearl. Generalized best-first search strategies and the optimality of A*. *Journal of the ACM (JACM)*, 32(3):505–536, 1985.

[Ds02] A. Dogan and F. sgnger. Matching and scheduling algorithms for minimizing execution time and failure probability of applications in heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(3):308–323, 2002.

[ECQ09] N. Eagle, A. Clauset, and J. Quinn. Cellular tower data for anticipatory computing. In *Proceedings of the AAAI 2009 Spring Symposium (AAAI'09)*, 2009.

[Epp98] D. Eppstein. Finding the k shortest paths. *SIAM Journal on Computing*, 28(2):652–673, 1998.

[EW97] M. M. Eshaghian and Y. C. Wu. Mapping heterogeneous task graphs onto heterogeneous system graphs. In *Sixth IEEE Heterogeneous Computing Workshop (HCW'97)*, page 147, 1997.

[FGBA96] A. Fox, S. D. Gribble, E. A. Brewer, and E. Amir. Adapting to network and client variability via on-demand dynamic distillation. In *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, 1996.

[FGCB98] A. Fox, S. D. Gribble, Y. Chawathe, and E. A. Brewer. Adapting to network and client variation using active proxies: Lessons and perspectives. *IEEE Personal Communications*, 5(4):10 – 19, 1998.

[Fra96] L. Franken. *Quality of Service Management: a Model-Based Approach*. PhD thesis, University of Twente, The Netherlands, 1996.

[GA90] A. Ghafoor and I. Ahmad. An efficient model of dynamic task scheduling for distributed systems. *Fourteenth Annual International Computer Software and Applications Conference (COMPSAC'90)*, pages 442–447, Oct-2 Nov 1990.

[GHB08] M. C. Gonzalez, C. A. Hidalgo, and A.-L. Barabasi. Understanding individual human mobility patterns. *Nature*, 453(7196):779–782, June 2008.

[GN06] X. Gu and K. Nahrstedt. Distributed multimedia service composition with statistical QoS assurances. *IEEE Transactions on Multimedia*, 8(1):141, 2006. 1520-9210.

[GNM+03] X. Gu, K. Nahrstedt, A. Messer, I. Greenberg, and D. Milojicic. Adaptive offloading inference for delivering applications in pervasive computing environments. *Proceedings of the First IEEE International Conference on Pervasive Computing and Communications (PerCom 2003)*, pages 107–114, March 2003.

[Gou99] K. M. Goudarzi. *Consistency Preserving Dynamic Reconfiguration of Distributed Systems*. PhD thesis, Imperial College, London, 1999.

[GTE07] Y. Gu, Y. Tian, and E. Ekici. Real-time multimedia processing in video sensor networks. *Signal Processing: Image Communication*, 22(3):237–251, 2007.

[HBL+98] R. Han, P. Bhagwat, R. LaMaire, T. Mummert, V. Perret, and J. Rubas. Dynamic adaptation in an image transcoding proxy for mobile web browsing. *IEEE Personal Communications [see also IEEE Wireless Communications]*, 5(6):8–17, 1998. 1070-9916.

[HL92] P. Hansen and K. W. Lih. Improved algorithms for partitioning paroblems in parallel, pipelined, and distributed computing (correspondence). *IEEE Transactions on Computers*, 41(6):769–771, 1992.

[HL95] B. Hendrickson and R. W. Leland. A multi-level algorithm for partitioning graphs. In *ACM/IEEE conference on Supercomputing*, 1995.

[HM05] J. Hu and R. Marculescu. Energy- and performance-aware mapping for regular NoC architectures. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 24(4):551–562, 2005. 0278-0070.

[HNR03] M. Hicks, A. Nagarajan, and R. V. Renesse. User-specified adaptive scheduling in a streaming media network. In *IEEE 6th International Conference on Open Architectures and Network Programming (OPENARCH)*, 2003.

[HP03] J. Hennessy and D. Patterson. *Computer Architecture - A Quantitative Approach*. Morgan Kaufmann, 2003.

[IB95] M. A. Iqbal and S. H. Bokhari. Efficient algorithms for a class of partitioning problems. *IEEE Transactions on Parallel and Distributed Systems*, 6(2):170, 1995. 1045-9219.

[IJZ04] R.S.H. Istepanian, E. Jovanov, and Y.T. Zhang. Guest editorial introduction to the special section on m-health: Beyond seamless mobility and global wireless health-care connectivity. *IEEE Transactions on Information Technology in Biomedicine*, 8(4):405–414, Dec. 2004.

[ILP05] R.S.H. Istepanian, S. Laxminarayan, and C.S. Pattichis. *M-Health: Emerging Mobile Health Systems*. Springer, 2005.

[JHM04] W. M. Johnston, J. R. P. Hanna, and R. J. Millar. Advances in dataflow programming languages. *ACM Computing Surveys*, 36(1):1–34, 2004.

[JHW+06] V. M. Jones, A. V. Halteren, I. Widya, N. Dokovsky, G. Koprinkov, R. Bults, D. Konstantas, and R. Herzog. Mobihealth: Mobile health services based on body area networks. In Robert S. H. Istepanian, S. Laxminarayan, and C.S. Pattichis, editors, *M-health Emerging Mobile Health Systems*, pages 219–236. Springer, 2006.

[JIT+06] V. M. Jones, F. Incardona, C. Tristram, S. Virtuoso, and A. Lymberis. Future challenges and recommendations. In Robert S. H. Istepanian, S. Laxminarayan, and C.S. Pattichis, editors, *M-health Emerging Mobile Health Systems*, pages 267–270. Springer, 2006.

[JMB+07] V.M. Jones, H. Mei, T. Broens, I. Widya, and J. Peuscher. Context aware body area networks for telemedicine. In *Pacific-Rim Conference on Multimedia (PCM'07)*, 2007.

[JW08] A. D. Jurik and A. C. Weaver. Remote medical monitoring. *IEEE Computer*, 41(4):96–99, 2008. 0018-9162.

[KA98] M. Kafil and I. Ahmad. Optimal task assignment in heterogeneous distributed computing systems. *IEEE Concurrency [see also IEEE Transactions on Parallel and Distributed Technology]*, 6(3):42, 1998. 1092-3063.

[KA99] Y. K. Kwok and I. Ahmed. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys*, 31(4), 1999.

[KHH05] M. Kuorilehto, M. Hännikäinen, and T. D. Hämäläinen. A survey of application distribution in wireless sensor networks. *EURASIP Journal on Wireless Communications and Networking*, 5(5):774–788, 2005.

[KK95] G. Karypis and V. Kumar. Multilevel graph partitioning schemes. In *24th International Conference on Parallel Processing (ICPP'95)*, pages 113–122. CRC Press, 1995.

[KK96] G. Karypis and V. Kumar. Parallel multilevel graph partitioning. In *10th International Parallel Processing Symposium (IPPS'96)*, pages 314–319, Washington, DC, USA, 1996. IEEE Computer Society.

[KLZ97] Y. Kopidakis, M. Lamari, and V. Zissimopoulos. On the task assignment problem: two new efficient heuristic algorithms. *Journal of Parallel and Distributed Computing*, 42(1):21–29, 1997.

[KM85] J. Kramer and J. Magee. Dynamic configuration for distributed systems. *IEEE Transactions on Software Engineering*, 11(4):424–436, 1985.

[KM90] J. Kramer and J. Magee. The evolving philosophers problem: dynamic change management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, Nov 1990.

[Lis87] B. Liskov. Keynote address - data abstraction and hierarchy. In *OOPSLA '87: Addendum to the proceedings on Object-oriented programming systems, languages and applications (Addendum)*, pages 17–34, New York, NY, USA, 1987. ACM.

[LL02] W. Y. Lum and F. C. M. Lau. A context-aware decision engine for content adaptation. *IEEE Pervasive Computing*, 1(3):41–49, 2002. 1536-1268.

[Lo88] V. M. Lo. Heuristic algorithms for task assignment in distributed systems. *IEEE Transactions on Computers*, 37(11):1384, 1988. 0018-9340.

[LP01] R. Litiu and A. Prakash. DACIA: a mobile component framework for building adaptive distributed applications. *ACM SIGOPS Operating Systems Review*, 35(2), 2001.

[LS97] C. H. Lee and K. G. Shin. Optimal task assignment in homogeneous networks. *IEEE Transactions on Parallel and Distributed Systems*, 8(2):119–129, 1997. 1045-9219.

[LV07] Y. Liu and B. Veeravalli. Heuristic-path and observer based low-energy scheduling algorithms for body area network systems. *IEEE Biomedical Circuits and Systems Conference*, pages 167–170, 2007.

[MBP+09a] H. Mei, B. J. Beijnum, P. Pawar, I. Widya, and H. Hermens. A*-based task assignment algorithm for context-aware mobile patient monitoring systems. In *15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2009)*, pages 245–254, Washington, DC, USA, 2009. IEEE Computer Society.

[MBP+09b] H. Mei, B. J. Beijnum, P. Pawar, I. Widya, and H. Hermens. Context-aware dynamic reconfiguration of mobile patient monitoring systems. In *4th IEEE International Symposium on Wireless Pervasive Computing (ISWPC 2009)*, 2009.

[MBW+08] H. Mei, B. J. Beijnum, I. Widya, V. Jones, and H. Hermens. Enhancing the performance of mobile healthcare systems based on task-redistribution. In *2nd IEEE Workshop on Mission Critical Networking (INFOCOM 2008 workshops)*, 2008.

[MCC04] Y. C. Ma, T. F. Chen, and C. P. Chung. Branch-and-bound task allocation with task clustering-based pruning. *Journal of Parallel and Distributed Computing*, 64(11):1223 – 1240, 2004.

[MFT00] J. K. Muppala, R. M. Fricks, and K. S. Trivedi. Techniques for dependability evaluation. In W. Grasman, editor, *Computational Probability*, pages 445–480. Kluwer Academic Publishers, 2000.

[Mit00] S. R. Mitchell. *Dynamic Configuration of Distributed Multi-media Components*. PhD thesis, University of London, 2000.

[MKK+05] A. Mandal, K. Kennedy, C. Koelbel, G. Marin, J. Mellor-Crummey, B. Liu, and L. Johnsson. Scheduling strategies for mapping application workflows onto the grid. In *IEEE International Symposium of the High Performance Distributed Computing (HPDC'05)*, pages 125–134, Washington, DC, USA, 2005. IEEE Computer Society.

[MPW07] H. Mei, P. Pawar, and I. Widya. Optimal assignment of a tree-structured context reasoning procedure onto a host-satellites system. pages 1–9, March 2007.

[MS98] M. Maheswaran and H. J. Siegel. A dynamic matching and scheduling algorithm for heterogeneous computing systems. In *Seventh IEEE Heterogeneous Computing Workshop. (HCW'98)*, 1998.

[MvBBW+09] H. Mei, van Beijnum B.J., I. Widya, V. Jones, and H. Hermens. Context-aware task redistribution for enhanced m-health application performance. In P. Olla and J. Tan, editors, *Mobile Health Solutions for Biomedical Applications*. Information Science Reference, 2009.

[MW07] H. Mei and I. Widya. Context-aware optimal assignment of a chain-like processing task onto chain-like resources in m-health. In *Proceedings of the 7th international conference on Computational Science, Part III (ICCS 2007)*, pages 424–431, Berlin, Heidelberg, 2007. Springer-Verlag.

[MWB+07] H. Mei, I. Widya, T. Broens, P. Pawar, A. van Halteren, B. Shishkov, and M. van Sinderen. A framework for smart distribution of bio-signal processing units in m-health. In *2nd International Conference on Software and Data Technologies (ICSOFT 2007)*, pages 249–255, 2007.

[NO91] D. M. Nicol and D. R. O'Hallaron. Improved algorithms for mapping pipelined and parallel computations. *IEEE Transactions on Computers*, 40(3):295–306, 1991.

[NSN+97] Brian D. Noble, M. Satyanarayanan, Dushyanth Narayanan, James Eric Tilton, Jason Flinn, and Kevin R. Walker. Agile application-aware adaptation for mobility. In *Sixteen ACM Symposium on Operating Systems Principles*, St. Malo, France, 1997.

[NT93] M. G. Norman and P. Thanisch. Models of machines and computation for mapping in multicomputers. *ACM Computing Surveys*, 25(3):263–302, 1993. 56 citations.

[OM95] B. Olstad and F. Manne. Efficient partitioning of sequences. *IEEE Transactions on Computers*, 44(11):1322–1326, 1995.

[OYL06] S. Ou, K. Yang, and A. Liotta. An adaptive multi-constraint partitioning algorithm for offloading in pervasive systems. *Fourth Annual IEEE International Conference on Pervasive Computing and Communications (PerCom 2006)*, March 2006.

[PA04] A. Pinar and C. Aykanat. Fast optimal load balancing algorithms for 1D partitioning. *Journal of Parallel and Distributed Computing*, 64(8):974–996, 2004.

[PLS+06] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer. Network-aware operator placement for stream-processing systems. In *22nd International Conference on Data Engineering (ICDE'06)*, 2006.

[PMW+07] P. Pawar, H. Mei, I. Widya, B. J. van Beijnum, and A. van Halteren. Context-aware task assignment in ubiquitous computing environment - a genetic algorithm based approach. pages 2695–2702, Sept. 2007.

[PvBMH09] P. Pawar, B.-J. van Beijnum, H. Mei, and H. Hermens. Towards proactive context-aware service selection in the geographically distributed remote patient monitoring system. In *4th International Symposium on Wireless Pervasive Computing (ISWPC 2009)*, Febuary 2009.

[RAC+02] M. J. Rutherford, K. M. Anderson, A. Carzaniga, D. Heimbigner, and A. L. Wolf. Reconfiguration in the enterprise javabean component model. In *The IFIP/ACM Working Conference on Component Deployment (CD'02)*, pages 67–81, London, UK, 2002. Springer-Verlag.

[RCD91] S. Ramakrishnan, I. H. Cho, and L. A. Dunning. A close look at task assignment in distributed systems. In *Tenth Annual Joint Conference of the IEEE Computer and Communications Societies. (INFOCOM '91)*, pages 806–812 vol.2, 1991.

[Rei96] J. Reid. *A Telemedicine Primer: Understanding the Issues*. Innovative Medical Communications, 1996.

[RN02] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2002.

[Rus92] S. Russell. Efficient memory-bounded search methods. In *10th European conference on Artificial intelligence (ECAI '92)*, pages 1–5, New York, NY, USA, 1992. John Wiley & Sons, Inc.

[RZ07] A. Rahmati and L. Zhong. Context-for-wireless: Context-sensitive energy-efficient wireless data transfer. In *5th international conference on Mobile systems, applications and services (Mobisys'07)*, pages 165–178, 2007.

[Sat04] M. Satyanarayanan. The many faces of adaptation. *IEEE Pervasive Computing*, 2004.

[SC90] J. P. Sheu and Z. F. Chiang. Efficient allocation of chain-like task on chain-like network computers. *Information Processing Letters*, 36(5):241 – 246, 1990.

[SCS+04] S. Shivle, R. Castain, H. J. Siegel, A. A. Maciejewski, T. Banka, K. Chindam, S. Dussinger, P. Pichumani, P. Satyasekaran, and W. Saylor. Static mapping of subtasks in a heterogeneous ad hoc grid environment. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS'04)*, 2004.

[Smi80] R.G. Smith. The contract net protocol: High-level communication and control in a distributed problem solver. *IEEE Transactions on Computers*, C-29(12):1104–1113, Dec. 1980.

[ST85] C. C. Shen and W. H. Tsai. A graph matching approach to optimal task assignment in distributed computing systems using a minimax criterion. *IEEE Transactions on Computers*, C-34(3):197–203, 1985.

[Sto77] H. S. Stone. Multiprocessor scheduling with the aid of network flow algorithms. *IEEE Transactions on Software Engineering*, 3:85–93, 1977.

[SW97] M. Stoer and F. Wagner. A simple min-cut algorithm. *Journal of the ACM*, 44(4):585–591, 1997.

[SWG92] S. M. Shatz, J. P. Wang, and M. Goto. Task allocation for maximizing reliability of distributed computer systems. *IEEE Transactions on Computers*, 41(9):1156–1168, 1992.

[TcZ07] Nesime Tatbul, Uğur Çetintemel, and Stan Zdonik. Staying fit: efficient load shedding techniques for distributed stream processing. In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, pages 159–170. VLDB Endowment, 2007.

[THVH06] T. Tonis, H. J. Hermens, and M. Vollenbroek-Hutten. Context aware algorithm for discriminating stress and physical activity versus epilepsy. Technical report, AWARENESS deliverables (D4.18), 2006.

[UAKI06] B. Ucar, C. Aykanat, K. Kaya, and M. Ikinci. Task assignment in heterogeneous computing systems. *Journal of Parallel and Distributed Computing*, 66(1), 2006.

[vSBM05] M. van Sinderen, H. Batteram, and E. Meeuwissen. Scope and scenarios. Technical report, AWARENESS deliverables (D1.1), 2005.

[War00] I. Warren. *A Model for Dynamic Configuration which Preserves Application Integrity*. PhD thesis, Lancaster University, UK, 2000.

[Weg03] M. Wegdam. *Dynamic Reconfiguration and Load Distribution in Component Middleware*. PhD thesis, University of Twente, The Netherlands, 2003.

[WHZ01] D. Wu, Y.T. Hou, and Y. Q. Zhang. Scalable video coding and transport over broadband wireless networks. *Proceedings of the IEEE*, 89(1):6–20, Jan 2001.

[Woe01] G. J. Woeginger. Assigning chain-like tasks to a chain-like network. In *Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*, pages 765 – 766, 2001.

[WS96] I. Warren and I. Sommerville. A model for dynamic configuration which preserves application integrity. *Third International Conference on Configurable Distributed Systems*, pages 81–88, 1996.

[WSRM97] L. Wang, H. J. Siegel, V. R. Roychowdhury, and A. A. Maciejewski. Task matching and scheduling in heterogeneous computing environments using a genetic-algorithm-based approach. *Journal of Parallel and Distributed Computing*, 47(1):8–22, 1997.

[WvHK06] K. E. Wac, A. T. van Halteren, and D. Konstantas. Qos-predictions service: infrastructural support for proactive qos- and context-aware mobile services. In *Proceedings of the International Workshop on Context-Aware Mobile Systems (CAMS)*, volume 4278, pages 1924–1933. Springer Verlag, October 2006.

[XQ08] T. Xie and X. Qin. An energy-delay tunable task allocation strategy for collaborative applications in networked embedded systems. *IEEE Transactions on Computers*, 57(3):329–343, 2008.

[XRC+02] Y. Xiao, J. Rosdahl, S. L. D. Center, M. Linear, and U. T. Draper. Throughput and delay limits of ieee 802.11. *IEEE Communications Letters*, 6(8):355–357, 2002.

[YC94] G. H. Young and C. L. Chan. Efficient algorithms for assigning chain-like tasks on a chain-like network computer. In *Proceedings of the 5th International Symposium on Algorithms and Computation*, pages 607 – 615, 1994.

[Yeh05] C. C. Yeh. On the power-aware resource allocation for linear-pipelined real-time tasks. In *19th International Conference on Advanced Information Networking and Applications*, 2005.

[YP05] Y. Yu and V. K. Prasanna. Energy-balanced task allocation for collaborative processing in wireless sensor networks. *Mobile Networks and Applications*, 10(1):115–131, 2005.

[ZL07] Z. Zhao and W. Li. Dynamic reconfiguration of distributed data flow systems. *31st Annual International on Computer Software and Applications Conference (COMPSAC 2007)*, 2:535–540, July 2007.

[ZWS+08] B. Zhao, M. Wang, Z. Shao, J. Cao, K. Chan, and J. Su. Topology aware task allocation and scheduling for real-time data fusion applications in networked embedded sensor systems. pages 293–302, Aug. 2008.

Abbreviations

BAN	Body Area Network
BSPU	Bio-Signal Processing Unit
CMN	Continuous Media Network
DAG	Directed Acyclic Graph
DWG	Doubly Weighted Graph
ECG	Electro-Cardio Gram
GA	Genetic Algorithm
GPRS	General Packet Radio Service
GPS	Global Positioning System
GSM	Global System for Mobile communications
HRI	Heart Rate Increase
ICT	Information and Communication Technology
ISA	Instruction Set Architecture
MADE	Monitoring, Analysis, Decision, Enforcement
MANET	Mobile Ad-hoc NETwork
MPMS	Mobile Patient Monitoring System
PDA	Personal Digital Assistant
PE	Processing Element
PROC	Processing and ROuting Component
QoS	Quality Of Service
RPC	Remote Procedure Call
SB	Sum-Bottleneck

SLP	Service Location Protocol
SOA	Service Oriented Architecture
SSB	Summation of S weight and B weight
TIG	Task Interaction Graph
TPG	Task Precedence Graph
WLAN	Wireless Local Area Network

About The Author

Hailiang was born in Beijing, China. He received his Bachelor of Science degree from the Beijing University of Technology in 2001 in the area of Electrical Engineering, and his Master of Science degree from the Technical University of Delft in 2003 in the same area. Between 2003 and 2005, he worked as a researcher in the System Architecture and Networking group at the Eindhoven University of Technology. He participated in the ITEA project Space4U where his focus was on the network protocol and security in remote device management. In 2005, he started working in the Architecture and Services of Network Applications group and the Telemedicine/BSS group at the University of Twente as a PhD candidate. His research focus is on the ICT technology that enables smart distribution of bio-signal process in mobile healthcare. His work is part of the Freeband AWARENESS project. He has been a reviewer of several international conferences and workshops and has served as a program committee member in the International Workshop on e-Health Services and Technologies in 2008 and 2009. He is now working at the Netherlands Bioinformatics Centre as a system integration engineer.

Below is a list of his publications that are relevant to this thesis:

- Hailiang Mei, Bert-Jan van Beijnum, Pravin Pawar, Ing Widya, and Hermie Hermens. A*-Based Task Assignment Algorithm for Context-Aware Mobile Patient Monitoring Systems. 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2009), August 2009, Beijing, China.
- Pravin Pawar, Bert-Jan van Beijnum, Hailiang Mei, and Hermie Hermens. Towards Proactive Context-Aware Service Selection in the Geographically Distributed Remote Patient Monitoring System. IEEE International Symposium on Wireless and Pervasive Computing (ISWPC 2009), February 2009, Melbourne, Australia.
- Hailiang Mei, Bert-Jan van Beijnum, Pravin Pawar, Ing Widya, and Hermie Hermens. Context-Aware Dynamic Reconfiguration of Mobile Patient Monitoring System. IEEE

International Symposium on Wireless and Pervasive Computing (ISWPC 2009), February 2009, Melbourne, Australia.

- Ing Widya, Hailiang Mei, Bert-Jan van Beijnum, Jacqueline Wijsman, and Hermie J. Hermens. Medical Information Representation Framework for Mobile Healthcare. In P. Olla and J. Tan, editors, Mobile Health Solutions for Biomedical Applications. Information Science Reference, 2009.
- Hailiang Mei, Bert-Jan van Beijnum, Ing Widya, Val Jones, and Hermie Hermens. Context-aware task redistribution for enhanced m-health application performance. In P. Olla and J. Tan, editors, Mobile Health Solutions for Biomedical Applications. Information Science Reference, 2009.
- Boris Shishkov, Hailiang Mei, Marten van Sinderen, and Thijs Tonis. Towards a NORM-driven design of context-aware E-Health applications. 2nd International workshop on e-Health Services and Technologies (part of ICSOFT 2008), July 2008, Porto, Portugal.
- Hailiang Mei, Bert-Jan van Beijnum, Ing Widya, Val Jones, and Hermie Hermens. Enhancing the Performance of Mobile Healthcare Systems Based on Task-Redistribution. 2nd IEEE Workshop on Mission Critical Networking (part of INFOCOM 2008), April 2008, Phoenix, US.
- Val Jones, Hailiang Mei, Tom Broens, Ing Widya, and Jan Peuscher. Context Aware Body Area Networks for Telemedicine. 8th Pacific-Rim Conference on Multimedia (PCM 2007), December 2007, Hong Kong, China.
- Hailiang Mei. Smart Distribution of Bio-Signal Processing Tasks in M-health. 4th OTM Academy Doctoral Consortium (part of OTM 2007), November 2007, Vilamoura, Algarve, Portugal.
- Pravin Pawar, Hailiang Mei, Ing Widya, Bert-Jan van Beijnum, and Aart van Halteren. Context-Aware Task Assignment in M-health - A Genetic Algorithm Based Approach. 2007 IEEE Congress on Evolutionary Computation (CEC 2007), September 2007, Singapore.
- Hailiang Mei, Ing Widya, Tom Broens, Pravin Pawar, Aart van Halteren, Boris Shishkov, and Marten van Sinderen. A Framework for Smart Distribution of Bio-signal Processing Units in M-health (invited paper). 2nd International Conference on Software and Data Technologies (ICSOFT 2007), July 2007, Barcelona, Spain.
- Hailiang Mei and Ing Widya. Context-Aware Optimal Assignment of a Chain-like Processing Task onto Chain-like Resources in M-Health. 7th International conference on computational science (ICCS 2007), May 2007, Beijing, China.
- Hailiang Mei, Pravin Pawar, and Ing Widya. Optimal Assignment of a Tree-Structured Context Reasoning Procedure onto a Host-Satellites System. 16th Heterogeneity in Computing Workshop (HCW 2007), March 2007, Long Beach, California.
- Hailiang Mei, Ing Widya, Aart van Halteren, and Bayu Erfianto. A Flexible Vital Sign Representation Framework for Mobile Healthcare. 1st International Conference on Pervasive Computing Technologies for Healthcare, December 2006, Innsbruck, Austria.