Learning Deployment Trade-offs for Self-Optimization of Internet of Things Applications

Arun kishore Ramakrishnan, Nayyab Zia Naqvi, Zubair Wadood Bhatti,
Davy Preuveneers and Yolande Berbers
iMinds-DistriNet, Department of Computer Science
KU Leuven
3001 Leuven, Belgium

Abstract

The Internet of Things (IoT) is the next big wave in computing characterized by large scale open ended heterogeneous network of things, with varying sensing, actuating, computing and communication capabilities. Compared to the traditional field of autonomic computing, the IoT is characterized by an open ended and highly dynamic ecosystem with variable workload and resource availability. These characteristics make it difficult to implement self-awareness capabilities for IoT to manage and optimize itself. In this work, we introduce a methodology to explore and learn the trade-offs of different deployment configurations to autonomously optimize the QoS and other quality attributes of IoT applications. Our experiments demonstrate that our proposed methodology can automate the efficient deployment of IoT applications in the presence of multiple optimization objectives and variable operational circumstances.

1 Introduction

No doubt, recent advances in ICT have changed our verve enormously. Out of many emerging technologies there is a continuous rise of highly distributed ambient computing environments such as the Internet of Things (IoT) and the Machine-to-Machine (M2M) communication paradigm. IoT is an open ended network infrastructure with self-configuring capabilities fueled by low cost wireless communication and efficient network performance. It is a dynamic network of uniquely identifiable fixed or mobile communicating *objects*. These objects collect data, relay information to one another, process the information collaboratively, and take actions in an autonomic way without human intervention. Smart homes and offices, smart health, assisted living, smart cities and transportation are only a few examples of possible application scenarios where IoT is playing a vital role. Also in this domain many significant self-* challenges exist. For example, one challenge on self-optimization is how to change the behavior of a system to achieve a desired functionality, while maintaining a balance with Quality of Service (QoS) and resource usage [21]. Self-optimization in the Internet of Things shifts the focus from design and deployment of a single or a few elements operating autonomously to a large complex ecosystem of a network of autonomous elements [16].

Most of the existing software platforms for IoT are highly domain-specific prohibiting seamless interoperability of *objects* across multiple vertical domains. The FP7 BUTLER project[1] aims to address this concern by achieving a secure, context-aware horizontal architecture for IoT by offering common functionality on three platforms - *Smart Object*, *Smart Mobile* and *Smart Server*. In this work we aim to predict and control the global system behavior resulting from self-optimization of the components deployed among these three different platforms. The dynamic deployment of software components in an IoT system has to take into account the resource characteristics of the application components and the platforms used for deployment in terms of processing power, bandwidth, battery life and connectivity [1]. Each platform has its own capabilities and limitations to achieve Quality of Service (QoS) requirements. The heterogeneity makes it more complex and challenging to cope with QoS requirements.

The main objective of our work is to find optimal distributed deployments and configurations of application components. We use annotated component graphs to model application compositions and Pareto-curves to represent the optimization options for each (type of) platform, i.e. the *Smart Object*, *Smart Mobile* and *Smart Server*. The resource optimization objectives are chosen with respect to the QoS requirements and the trade-offs on the computation vs. communication cost-benefits. For the runtime (re)configuration and (re)deployment,

[1] http://www.iot-butler.eu/

we use Markov Decision Processes to achieve the self-optimization capabilities of the system.

After discussing related work in section 2, we present some motivating use cases in the healthcare and wellbeing domain in section 3 from which we elicit relevant functional and non-functional requirements. We briefly outline our self-optimization approach in section 4. It is based on an offline exploration phase to collect relevant profiling information for optimization before actual deployment, and a runtime phase to autonomously adapt the deployment and configuration towards changing operational circumstances. In section 5 we evaluate the deployment and optimization trade-offs in our work, and finally conclude this paper with possible directions for future work in section 6.

2 Related work

The autonomic computing paradigm has been around for almost a decade with a primary vision of computing systems that can manage themselves [10, 8]. This vision is now gaining inroads into the Internet of Things (IoT), with many typical optimization criteria:

- increase the performance by deploying heavyweight application components on faster hardware.
- reduce the amount of communication and network latencies between distributed components.
- optimize the overall energy consumption of the application components on the different platforms.

Utility functions are often used to achieve self-optimization in distributed autonomic computing systems, both for the initial deployment of an application and its dynamic reconfiguration. Tesauro et al. [19] explored utility functions as a way to enable a collection of autonomic elements to continually optimize the use of computational resources in a dynamic, heterogeneous environment. Later work by Deb et al. [5] investigated how utility functions can be used to achieve self-optimized deployment of computationally intensive scientific and engineering applications in highly dynamic and large-scale distributed computing environments. Utility functions have also found their way into the cloud computing space [7, 11] where they are used to manage virtualized computational and storage resources that can scale on demand.

The problem with utility functions is that their definitions require a fair amount of domain-specific knowledge to be effective. To address this challenge, reinforcement learning is often considered to automatically infer optimal deployment strategies. Tesauro [17, 18] explored reinforcement learning for an online resource allocation task in a distributed multi-application computing environment with independent time-varying load in each application. Similar work was proposed by Vengerov [20] using reinforcement learning in conjunction with fuzzy rulebases to achieve the desired objective. However, long training times is a reoccurring concern that often outweighs the potential benefits of reinforcement learning.

Organic Computing is another paradigm that focuses on distributed systems that exhibit self-* properties. In [3], a generic observer/controller architecture is proposed to introduce self-organization in complex systems such as traffic light controllers. The observer collects relevant data, pre-processes and analyzes it to discover patterns which might affect the performance of the system. The controller explores the parameter space to discover settings that would suit the future states of the system, but also matches the appropriate parameter settings to the current state of the system. For the traffic controller use-case, an evolutionary algorithm-based approach is used to explore and optimize the solution space and discover appropriate parameter settings. The controller then compares the performance of the discovered parameter settings in a simulation environment and deploys the most appropriate setting at runtime.

Similarly, in [15] the authors propose a new framework for self-organizing systems, albeit for improving the efficiency in terms of functional requirements of the system. In line with the observer/controller architecture proposed in [3], an advisor (a high-level agent) monitors the performance of other agents in a distributed environment and provides suggestions to improve their performance. The main focus of the paper is to improve the overall efficiency of the system considering the openness and autonomy of the system along with low observability and controllability of the agents (such as in the domain of pick-up and delivery). The advisor gathers data, analyzes and extracts recurring tasks and optimizes the solutions for those recurring tasks. In the aforementioned use case, exception rules are generated based on the current environmental conditions in order to improve the efficiency of the pick-up/delivery systems.

The focus of both the papers [3, 15] is on optimizing the functionality of the system while considering scalability and robustness requirements of the system. Contrary to our approach, the optimal system configuration for the architecture in [3] is completely determined online. Such an approach may require considerable resources at runtime and hamper the feasibility on resource constrained devices. Although [15] relaxes the need of continuous monitoring by providing some autonomy to the application for a limited amount of time, it does not address the performance/efficiency trade-offs which is of utmost importance in resource constrained IoT systems.

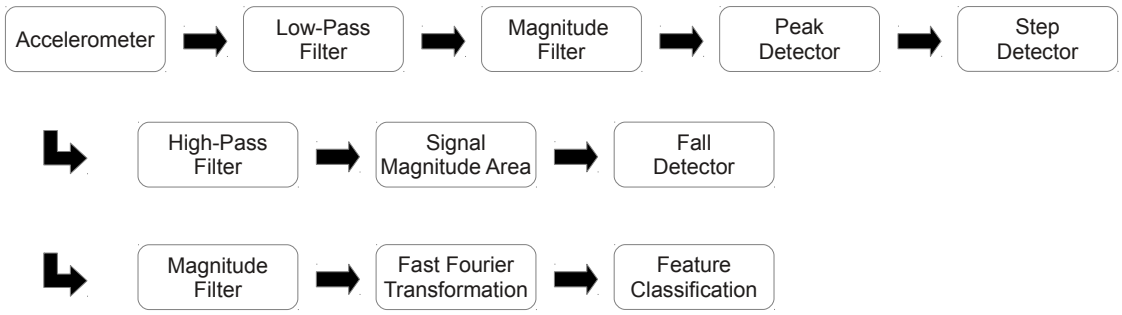

Figure 1: Component-based composition of the activity recognition application

Given the aforementioned optimization criteria, efficient deployment of application components in an IoT environment is often a multi-objective optimization problem [6, 13]. Note that these optimization objectives may conflict with one another (e.g. performance vs. energy consumption). In such cases, there does not exist a single solution that simultaneously optimizes each objective and resource trade-offs are to be made [9]. Pareto optimization [4, 22] is a technique that identities a set of Pareto-optimal solutions involving more than one objective function to be optimized simultaneously. We say that a solution − i.e. an allocation of resources − is Pareto-optimal if there exists no other alternative that would improve upon one objective function without deteriorating in at least one of the other objective functions.

On the one hand, the problem with utility functions (or optimization objectives) and Pareto-optimal solutions is that the Internet of Things is an open ended ecosystem of heterogeneous resources, making the crisp definition of Pareto-optimal solutions difficult due to an incomplete view on the external factors and uncertain circumstances that might influence the optimality. On the other hand, the applicability of the above learning approaches in an Internet of Things environment is usually hampered by the time and computational resources required to find a feasible or better solution. To address this concern, we aim to explore the feasibility of finding reasonable results in a reasonable amount of time by combining Pareto-optimization with reinforcement learning.

3 Scenarios and requirements for wellness and independent living

In this section, we will use some motivating scenarios from the healthcare and wellness domain as prototypical examples of IoT applications, and derive functional and non-functional requirements.

3.1 Use cases and components

Analysis of physical fitness and several health monitoring techniques revolve around the inference and prediction of human behavior. Accelerometer sensor data helps to analyze the human behavior in an effective way [14, 12]. We have implemented a variety of processing components in a modular fashion to enable a flexible deployment composition on the following platforms:

- *Smart Object*: Small appliances, sensors or actuators with limited computational power, storage capacity, communication capability, energy supply and primitive user interface are categorized as smart objects (e.g. RFID tagged objects, motion detectors, heating regulators).

- *Smart Mobile*: Devices with multi-modal user interfaces to enable user mobility through remote services are categorized as smart mobiles (e.g. smart phones, smart TVs). They usually have better resource provisions than smart objects.

- *Smart Server*: The aggregation and complex analysis of data from smart objects and smart mobiles are realized as services on smart servers (e.g. a local server or remote cloud computing set-up).

3.1.1 Use case 1 - motion activity recognition

In our first use case, we monitor the physical activity of the user by learning and classifying the activity of the user (e.g. standing, walking, running). We track the number of steps taken each day as a measure for wellbeing, and use it as input to classify higher levels of activity (e.g. cooking, watching TV, presenting at a meeting).

3.1.2 Use case 2 - fall detection for elderly

Another important parameter that characterizes the quality of independent life is the safety of the users in their

own homes. Ageing can affect all domains of life leading to physical infirmity and loss of mental or cognitive abilities necessitating safety monitoring applications. Our second use case specifically focuses on fall detection as a common safety monitoring application within an Ambient Assisted Living (AAL) environment.

3.1.3 Application components

Both use cases leverage a tri-axial accelerometer, a common mobile embedded inertial sensor found in most smartphones, but rely on different sampling rates and processing algorithms. A conceptual overview of the software components is provided in Figure 1, with an explanation of some of them below.

- **Accelerometer**: It produces a continuous stream of X,Y,Z acceleration data by sampling the sensor at a certain rate (see Figure 2).
- **Low-pass filter**: For mobility tracking we are interested in acceleration peaks that arrive at a frequency of maximum 5Hz (i.e. max 5 steps per second). We use the 'moving average' as a simple low-pass filter to remove high-frequency noise (see Figure 2).

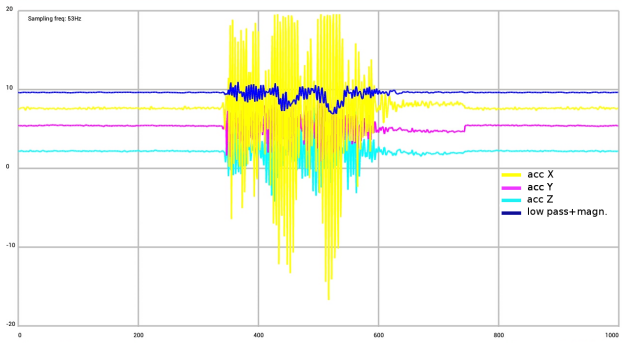

Figure 2: Accelerometer data and magnitude of signal after low-pass filter

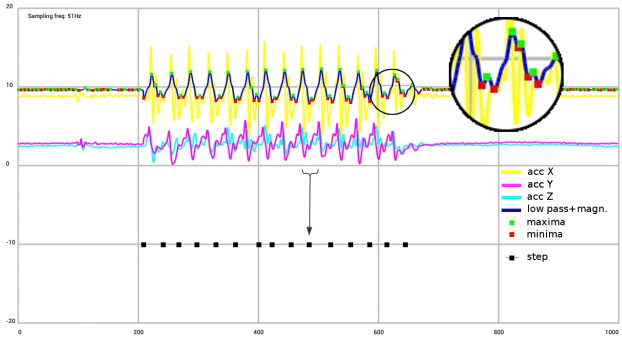

Figure 3: Peaks in magnitude signal and detected steps

- **Magnitude filter**: The orientation of the sensor is subject to change while moving around. Therefore, we carry out the signal analysis on the overall magnitude of the acceleration signal (see Figure 2).
- **Peak filter**: A single step is characterized by a pattern of several maxima and minima in the time domain of the acceleration signal. This component extracts these features in the signal for further analysis (see Figure 3).
- **Step detector**: It identifies the correct maxima/minima to correctly count the number of steps and to differentiate between standing still, walking and running (i.e. the peak rate) (see Figure 3).

Although this application is still fairly small in size and number of components, it manifests some interesting properties in the sense that the computational demands of certain components (e.g. the peak filter and step detector components) vary depending on the actual motion behavior of the user.

3.2 Requirements

The major (high-level) functional and non-functional requirements can be summarized as follows:

1. The system should be able to capture and store relevant sensor data and context information of the user to model, *learn, classify and predict the physical activity* of the users.
2. The system should have modular building blocks for data processing and activity recognition *on all three platforms for flexible distributed deployment*.
3. The deployment and configuration of the application components must *be adaptive at runtime to optimize for performance, latency, network communication (or QoS in general)*.

For example, delaying or offloading the accelerometer data processing will help to optimize the autonomy of battery powered sensors or mobiles.

Many opportunities for optimization may exist, i.e. different distributed deployments of the application components and different configurations per component. The challenge is to find and analyze the different optimization trade-offs in an open ended and dynamic IoT ecosystem of *Smart Objects*, *Smart Mobiles* and *Smart Servers*, each with varying sensing, communication, computation and storage capabilities.

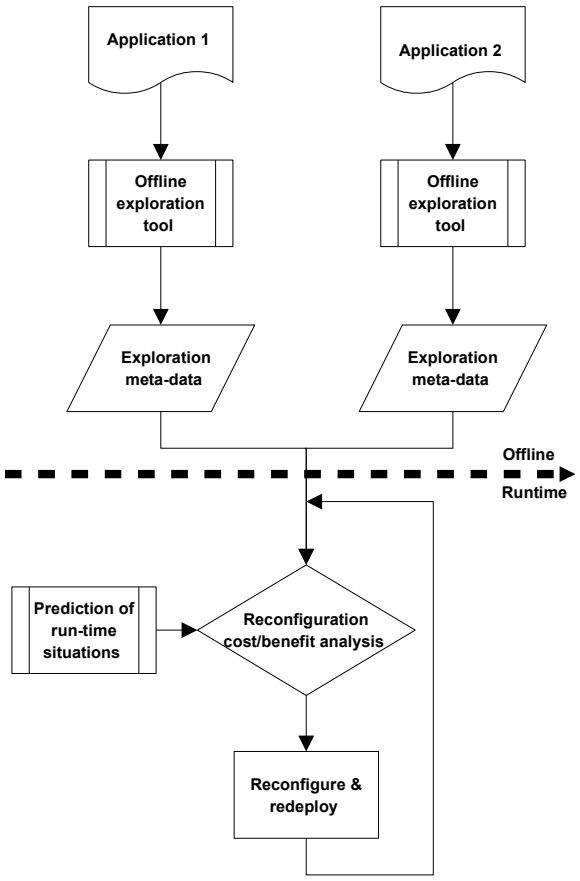

Figure 4: Overview of the self-optimization approach illustrating the offline and runtime phases

4 Conceptual overview of the deployment and optimization methodology

It is impossible to determine in advance where every component will run due to the dynamic interaction of these devices with the environment and the user. The multitude of parameters associated with the various possible configurations under varying workload and resource availability makes it almost impossible to manually finetune the components for best overall system performance, necessitating the introduction of self-* properties in IoT applications. Performing detailed cost benefit analysis for self-management decisions from scratch at runtime causes a large overhead. We reduce this overhead by balancing the offline and runtime efforts of making these decisions.

Our overall approach is based on an offline exploration phase to collect relevant profiling information for optimization before actual deployment, and a runtime phase to autonomously adapt the deployment and configuration towards changing operational circumstances. An overview of the approach is given in Figure 4.

4.1 Offline exploration of deployment and configuration options

Figure 5 gives an overview of the offline exploration for the preprocessing of deployment and configuration decisions. The component-based application is first profiled to obtain an annotated component graph. This annotated component graph is used for the exploration of the Pareto-optimal deployments and configurations and a reconfiguration cost matrix is constructed only for Pareto-optimal configurations. The runtime system uses the explored Pareto-optimal configurations and the reconfiguration matrices in order to make self-optimization decisions at runtime.

4.1.1 Deriving the annotated component graph

We use annotated component graphs as a high level model of computation to represent the application in order to explore the trade-offs between the different deployment configurations of the application. An annotated component graph is a directed graph where the nodes represent the components of an application, and the edges represent the data flow between the components. These nodes and edges are annotated with metadata representing the hard constraints, costs and resource requirements of the components.

Let us again consider the step counting application as an example. Some components of the application may be deployed on different platforms, i.e. a *Smart Object*, *Smart Mobile* and *Smart Server*. In order to generate an annotated component graph for this application the following steps are carried out:

1. Use the component model of the application and identity the data flows (similar to the one shown in Figure 1). The data flow graph acts as skeleton for the annotated component graph.
2. Instrument the communication interfaces of components to measure the amount of data transferred between components.
3. Run every component of the application on all the different platforms possible, profiling its execution time, energy consumption and data transferred between components, each time.
4. Calculate the memory requirements of every component by monitoring the changes in stack and heap sizes, as components are added and removed from the platform.
5. Repeat steps 3 and 4 over a range of component configurations (e.g. a different sampling rate) and/or simulated inputs (e.g. accelerometer traces of different activities and individuals).

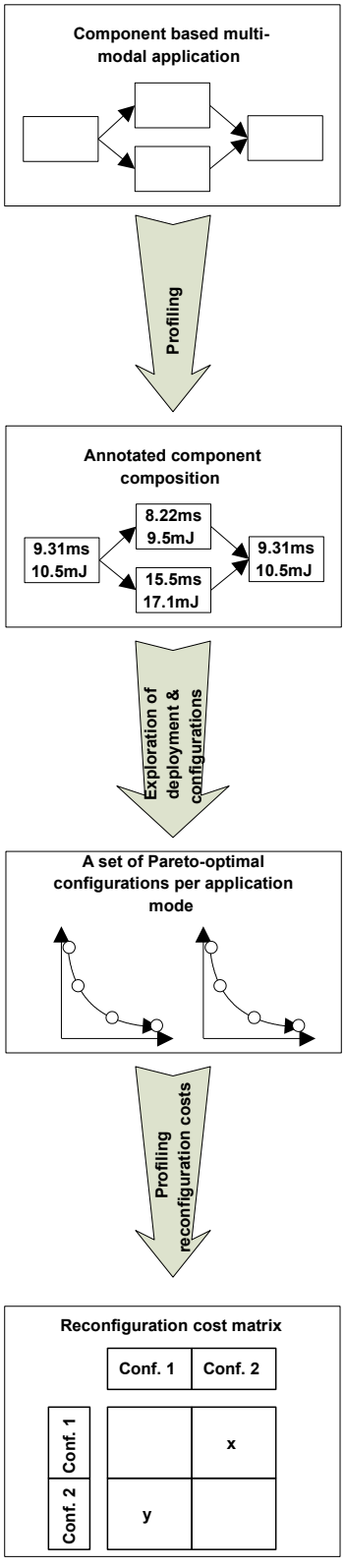

Figure 5: Overview of the offline exploration phase

The only hard constraint for this application is that the accelerometer component can only execute on devices with such a sensor. Adding all this meta-data to the data flow graph generates the annotated component graph of the application and we use it as an intermediate model for exploring deployment trade-offs at design time.

Reconfigurable components: Some components have configuration options that affect their resource costs and requirements. For example, lowering the accelerometer sampling rate from 50Hz to 15Hz decreases the CPU time, communication and energy consumption of the activity recognition components, but increases the recognition error rate. For such components we annotate the component graph with metadata for a discretized range of parameter options, i.e. the CPU time and energy consumption values for the supported sampling rates.

Variability: Some components have stochastic non-functional performance properties (see Figure 6). For example, the communication throughput of a wireless node could be affected by external factors (e.g. interference). To define the Pareto-fronts (or Pareto-curves) one usually takes the worst case execution values after profiling to define the Pareto-points. Given that the IoT ecosystem is quite heterogeneous and open ended in nature, pursuing such a pessimistic approach will easily lead to undesirable solutions. Therefore, we define the Pareto-points based on the most likely execution values. However, to still be able to assess the impact of a worst case execution scenario for a particular deployment and configuration (i.e. a specific Pareto-point), we incorporate the likelihood distribution of the profiled execution values in each Pareto-point leading to a Pareto-front (i.e. a set of Pareto-optimal solutions) with some degree of variability.

4.1.2 Exploring the Pareto-optimal trade-offs

We model the problem of deploying an application to a heterogeneous network of self-managing *Smart Objects*, *Smart Mobiles* and *Smart Servers* as a constraint-based optimization problem and use a CPLEX based solver to explore the Pareto-optimal set of solutions. The details of expressing software deployment on hardware resources are described in our previous work [2].

In a Pareto-optimal set of solutions, every solution is better than all other solutions according to at least one functional or non-functional criterion. For example, Table 1 refers to a scenario of fancy and cheap hotels close to the beach. Hotels A, E and F can be eliminated because they are not Pareto-optimal. Also note that Hotel D is not the best in any optimization objective (stars, distance to beach and price), but it is Pareto-optimal. Although we are mainly interested in activity recognition

Hotel	Stars	Distance to beach	Price
A	**	0.7	80
B	*	0.2	40
C	***	1.3	100
D	**	0.3	70
E	**	0.5	90
F	**	1.5	120

Table 1: Maximization problem with multiple optimization criteria

as a motivating scenario, we use this example to offer a better understanding of Pareto-curves with multiple optimization criteria.

In our approach, eliminating deployment and configuration options that are not Pareto-optimal reduces the search space for the runtime reconfiguration decision from all possible configurations to the set of Pareto-optimal configurations. For example, consider the step counting application which consists of 5 components (see Figure 1) with a fairly simple pipe-and-filter architectural style. Assume we aim to deploy this application composition in a distributed setting on a simple sensor platform with limited processing capabilities (i.e. a *Smart Object*) and on a resource rich platform (i.e. a *Smart Server*). The deployment decision then boils down to figuring out which components are deployed on the sensor and which ones are deployed on the server. The only hard constraint for the deployment of this application is that the *Accelerometer* component must be deployed on the sensor platform. The other components can be deployed on either side, theoretically leading to 16 different deployment configurations of which a subset are Pareto-optimal. Obviously, extreme deployment configurations where components 1, 3 and 5 are on the sensor and components 2 and 4 are on the server will never be optimal due to the high communication cost.

In order to explore multi-dimensional Pareto-optimal surfaces, the problem is modeled using of parameterizable constraints. These parameters are then iteratively varied over a discretized range, invoking the solver each time to find a point on a Pareto surface. For example, an energy consumption versus Quality of Service Pareto curve is explored for the step counting algorithm by iteratively finding minimum energy solutions for different QoS constraints. It is important to note that there are no dependencies among the different invocations of the CPLEX solver. While finding solutions for this application takes several minutes on a single machine (depending on how many simulations are carried out), we can speed up this process by initiating parallel invocations of the CPLEX solver on a cluster of machines. This guarantees the feasibility of the approach for larger applications with many more configuration alternatives.

4.1.3 Reconfiguration cost matrix

A reconfiguration cost matrix is constructed by profiling the costs of reconfigurations and redeployments of components. For example, the cost of activation/deactivation of a component, establishing a local/remote component -to-component communication channel and transferring the state of an active component over a communication network. The size of this matrix is $O(N^2)$ where N is the number of possible configurations. As N can be become large, only the Pareto-optimal configurations are considered for reconfigurations.

4.2 Managing variability with runtime redeployment and reconfiguration

Traditionally, profiling of the application components is done with the assumption that each component will correspond to just one point in the Pareto search space. The openness in IoT can potentially create a lot of variability in the operational conditions of smart applications which in turn causes inconsistency in resource consumptions w.r.t. the Pareto-optimal solutions. For example, external environmental parameters such as network connectivity and communication bandwidth availability can vary depending on the living environment of the user. This operational variability makes it difficult to profile components in general. Similarly, the performance of an application component can vary depending on the user behavior. For example, the computational load of the *Step Detector* component (see Figure 1) will be different when the user is standing still (little processing due to no significant peaks in the accelerometer data) or walking (several peaks per second).

Rather than profiling application components as single points in the Pareto search space, we represent each application component with value distributions (through multiple profiling iterations) for systems where this variability in operational conditions is highly anticipated (as is the case for IoT systems). Each component is rep-

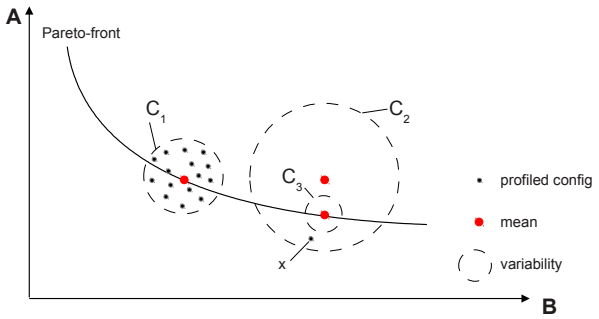

Figure 6: Variability in the profiled configurations

resented by statistical properties (e.g. min, max, median, average, standard deviation) discovered through multiple profiling iterations of the component under varying conditions. The example in Table 1 augmented with variable pricing (depending on season or room type) would require a similar Pareto-front representation.

We extended traditional Pareto-optimization methods to find a set of Pareto-optimal points taking into consideration this variability (see Figure 6). This basically means that the Pareto-optimal set not only considers configurations that are Pareto-optimal in the most likely scenarios (e.g. using the mean value of the optimization objectives), but also configurations that might be Pareto-optimal in less likely scenarios (e.g. min or max value of the optimization objectives). Assume in Figure 6 we aim to minimize for both objectives A and B. Configuration C_3 would be Pareto-optimal w.r.t. C_2 when considering the mean value of objective A (and the equal mean value for objective B). However, due to the difference in variability there might be a profiled configuration x of C_3 that has a lower value for objective A than any profiled configuration of C_2. Depending on which statistical property is chosen, we find different Pareto-optimal sets. In our approach, we take the union of these sets and refer to it as the *relaxed set of Pareto-optimal solutions* for any given statistical property. At runtime, we start off with a default statistical property to define the Pareto-optimal solutions, and propose using online reinforcement learning to discover whether the given context gives rise to other Pareto-optimal solutions that emerge in less likely situations. The major benefits of our approach are:

- Reduction of the (re)configuration search space by limiting the relevant working configurations to Pareto-optimal solutions.
- A modified Pareto-optimization method defining a relaxed set of Pareto-optimal solutions to handle the variability in the IoT working conditions.
- Finetuning the configuration at runtime by narrowing down the operational variability through reinforcement learning.

4.3 Analyzing cost/benefit trade-offs with Markov Decision Processes

With the relaxed set of Pareto-optimal solutions, we can find configurations that are optimal in a particular context. Whether these configurations remain beneficial over a certain time period is something we cannot infer from the Pareto-fronts.

Let us consider the 5 components in the step counting application in Figure 1 and the different deployment configurations. Whether any of these configurations remains optimal over time is unpredictable, and cannot be derived just from the offline generated Pareto-fronts. For example, the default sampling frequency for counting steps is set to 50Hz. However, if the system knows the person is not moving (e.g. sitting down in a meeting), it can reduce resource consumption by changing the configuration of the *Accelerometer* component and setting the sampling frequency to 15Hz. In this mode, it can detect a change in movement, and if so, set the sampling frequency back to 50Hz to start counting steps again.

We therefore model the relaxed Pareto-optimal configurations as states of a Markov Decision Process (MDP) along with the associated set of actions and rewards and find out the best possible (re)configuration policy over a finite time period. This uncertainty in potential benefits over time is introduced by a changing context in the operating environment of the system. Also note that these reconfigurations have associated reconfiguration costs which in turn would require the system to maintain the new configuration for a certain time T_{be} (break-even time) before it is actually able to benefit from deploying the new configuration.

As typical user activities are characterized by certain events that happen over certain period of time, the states are not expected to change at each time step. State transitions will be guided by transition rates, i.e. how quickly a transition takes place instead of how likely transitions are at each time step. Accordingly, a continuous time Markov process is ideal to model this problem but in order to reduce the complexity of the proposed system we have decided to utilize a discrete finite horizon MDP instead of a continuous MDP.

A classic discrete MDP is represented by a 4-tuple $S, A, P(s,s'), R(s,s')$ where S is the set of states, A is a super-set of sets of actions possible in each state, $P(s,s')$ is the transition probability between states s and s', and $R(s,s')$ is the reward for moving from state s to s'. The goal here is to discover and learn the expected rewards and best possible policy considering the transitions between configurations due to a changing context. The different parameters of our proposed MDP model are:

- **States**: a set of Pareto-optimal configurations for each application that can be possibly deployed in the system. It is represented as an n-tuple where n represents the number of platforms. If there are m possible configurations for each of the platforms, then the number of possible states is (m^n). Also, note that the momentarily Pareto-optimal global configurations are a small subset of these states.
- **Actions**: a set of possible state transitions that are allowed for optimal resource consumption are modeled as actions for each state, i.e. $a(s)$. We assume that all the application components have to be run on one of the available platforms.

- **Transition probability**: the probability with which state changes are anticipated in the system is called the transition probability. User activities or changing living conditions introduce randomness and cause deviations in the desired state transitions for the most likely Pareto-optimal solution.

- **Reward function**: the reward value is defined in terms of the resource consumption of the current and optimal configuration, and the time the optimal configuration will be active:

$$\frac{resource\ consumption_{opt.\ conf.}}{resource\ consumption_{curr.\ conf.}} * T$$

The resource consumption of a configuration on a particular platform is a weighted average of the m resources ρ_i involved:

$$resource\ consumption = \sum w_i * \rho_i \qquad 0 < i < m$$

We also assume that $T > T_{be}$, meaning that the cost incurred by the reconfiguration to the optimal one will be accounted for by running this new optimal configuration longer than the break even time T_{be}.

The optimal configuration is the one which maximizes the value η defined as follows:

$$\eta = \frac{\delta}{\sum w_{i,j} * \rho_{i,j}} \qquad 0 < i < m, 0 < j < n$$

where,

δ = accuracy of the step detector
$\rho_{i,j}$ = resources consumed (in %)
i = type of resource (memory, CPU, etc.)
j = type of platform (smart object, mobile or server)
w = weight to prioritize importance of resource

The above parameters can be explored offline to identify an optimal configuration for a given set of resources whose importance can be balanced by the user (e.g. battery and performance). However, the variability in the execution makes it impossible to guarantee that the aforementioned optimal solution will remain the same under all circumstances. We therefore use learning techniques to better identify the optimal deployment for the given operational circumstances of the application. The learning we propose is guided by an ε-greedy algorithm:

$$Q(s_{t+1}, a_{t+1}) = \varepsilon.mean(Q(s,a)) + (1-\varepsilon).max(Q(s,a))$$

where the first term helps the system to explore the relaxed Pareto-optimal configuration space and the second term exploits the learned best policy available at the time.

4.4 Discussion

Despite pre-optimizing the deployment decision of the IoT application components and implementing application specific optimization logic, the global optimal configuration cannot be determined during the offline exploration phase as it is dependent on multiple time-varying variables such as, user profile (e.g., age and other factors can influence how active a person is) preference of the user (e.g., to minimize computational load or communication bandwidth) and the operating environment (e.g., signal strength of the WiFi network). Hence in this paper, a smart adapter meta-component is proposed which implements a global runtime learner to drive the IoT application towards optimal configuration over time for any given user or operating conditions. The local application specific optimization logic (e.g., lowering the sampling frequency when the user is not active) takes the current (or aggregated) prediction of the application (e.g., the user is idle or active) as input and output the optimal configuration for the system. A simple version can be implemented by a look-up table with pre-defined output events and corresponding optimal configurations. Whereas the smart adapter is more generic, it takes current configuration of the IoT application and corresponding resource consumption in multiple platforms as input and recomputes the throughput of the predefined configurations which in turn is used by the reinforcement learning algorithm to learn the best policy for the user and associated operating environment. Given that the learned optimal configuration policy is tailored for the user, it will overrule the policies determined by the offline exploration phase. As the resource needs of the smart adapter is pre-determinable (due to its fixed introspection frequency), it is modeled by the reconfiguration matrix and the associated cost is considered in the overall efficiency of the application.

5 Experimental evaluation

We will demonstrate the feasibility of our approach with use case 1 (the step counting application as shown in Figure 1), and evaluate the proposed methodologies using the 5 components. This simple deployment scenario allows different deployment compositions on three different platforms *Smart Object*, *Smart Mobile* or *Smart Server*. For the sake of simplicity, we will only use two platforms in our experiments (see Figure 7):

Smart Object: We use a SunSPOT development board[2] with a 400MHz ARM 926ej-S processor with 1MB RAM and 8MB flash memories. The processor runs applications on top of a Java "Squawk" virtual machine.

[2] http://www.sunspotworld.com/docs/Yellow/eSPOT8ds.pdf

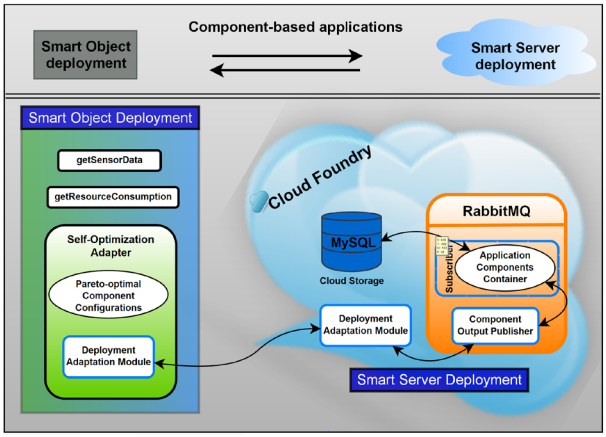

Figure 7: Experimental setup

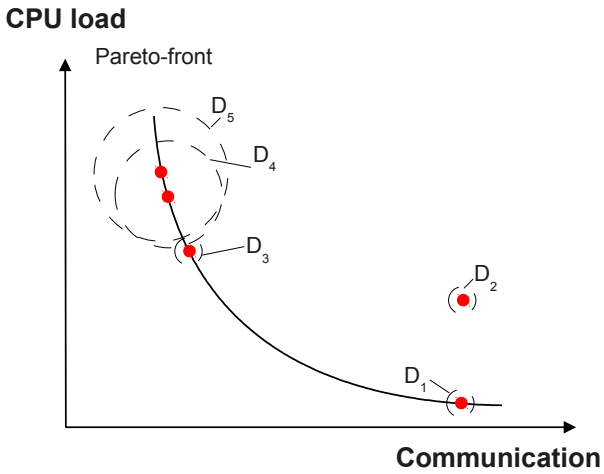

Figure 8: CPU load and network communication deployment trade-offs on the sensor

Component	CPU load	Communication
Accelerometer	8.09 ± 1.3 ms	5.5 ± 0.0 kB/sec
Low-pass filter	57.9 ± 2.1 ms	5.5 ± 0.0 kB/sec
Magnitude filter	18.2 ± 1.5 ms	1.8 ± 0.0 kB/sec
Peak detector	14.9 ± 9.7 ms	0.5 ± 0.4 kB/sec
Step detector	5.12 ± 4.8 ms	0.1 ± 0.1 kB/sec

Table 2: Performance benchmark of the individual components on the sensor

The board has an integrated IEEE 802.15.4 compliant Radio Transceiver CC2420 from Texas Instruments.

Smart Server: Our Smart Server infrastructure runs VMware's open source Platform-as-a-Service (PaaS) offering known as Cloud Foundry on a server with 8GB of memory and an Intel i5-2400 3.1GHz running a 64-bit edition of Ubuntu Linux 12.04. Cloud Foundry provides messaging and database servers as built-in services. We deployed its open source distribution, i.e. VCAP[3]. VCAP supports the AMQP-based *RabbitMQ*[4] server for messaging and *MySQL* for storage and persistence. All of the configuration is done in *Spring*, an application development framework. Finally, we exposed our loosely coupled application components as services, integrating *Apache CXF* with the Spring framework.

We profile the step counting components under different deployment and configuration scenarios with an objective to optimize the CPU load and the network communication costs. The results of the profiling on the sensor are shown in Table 2. Note that for the *Accelerometer, Low-pass filter* and *Magnitude filter* components there is little to no communication variability because the amount of data output is fixed and depending on the sampling rate of the accelerometer.

Given the fact that the deployment of the *Accelerometer* component is fixed, we have 16 different deployment options for the 4 remaining components. Some of the Pareto-optimal deployment options are (see Figure 8):

- D_1: Minimal computation on the sensor by having the *Accelerometer* component on the sensor and the 4 remaining sensor data processing components deployed on the server.
- D_2: The *Accelerometer* and *Low-pass filter* components deployed on the sensor and the other components on the server.
- D_3: The *Accelerometer*, *Low-pass filter* and *Magnitude filter* components deployed on the sensor and the other components on the server.
- D_4: All components except the *Step Detector* component deployed on the sensor.
- D_5: Highest CPU load on the sensor by having all the components deployed on the sensor and no communication cost between the sensor and the server.

Note that each deployment D_x represents the joint resource consumption and variability of the components deployed on the sensor. Examples of non-Pareto-optimal solutions include a.o. a deployment with the *Low-Pass Filter* and *Peak Detector* components on the server and the *Accelerometer*, *Magnitude Filter* and *Step Detector* components on the sensor. This mixed deployment causes a high communication cost.

We have also Pareto-fronts specifically for component reconfigurations. For example, the *Accelerometer* component can sample data at different rates, causing different CPU loads and communication throughput. Figure 8

[3]https://github.com/cloudfoundry/vcap
[4]http://www.rabbitmq.com

shows the results of sampling at 50Hz, whereas a profiling at 15Hz produces a similar deployment trade-off but with an overall lower CPU load and network communication. The fall detection use case requires a 100Hz sampling rate, but involves different components with corresponding Pareto-fronts.

5.1 Resource-driven deployment trade-offs

In a first experiment, we tested the automatic deployment of our application components with an initial deployment D_1. The optimization policy was set to reduce the energy consumption, which automatically triggered the deployment of all the components except the *Step Detector* component on the sensor (configuration D_4). We then changed the optimization policy to minimize network communication (cfr. a GPRS communication scenario that incurs a real financial cost). At this point, the deployment of the latter component was also moved to the sensor (configuration D_5).

5.2 Contextual configuration trade-offs

In a second experiment with periods without motion activity, the system learned that the stationary state of the individual would last for at least 10 minutes. In this state, there were no more peaks detected leaving the *Step Detector* component idle. This exceptional circumstance (i.e. no communication to this component) triggered the component to be deployed again on the server (configuration D_4), lowering the sampling frequency to 15Hz, and switching to the corresponding Pareto-front.

5.3 Learning self-optimization trade-offs

The effect of the *Peak Detector* in the Pareto-search space is more fuzzy compared to the three previous components in the processing chain (whose CPU load and communication variability is low). The variability in the resource consumption of the *Peak Detector* component is due to external factors. For example, for elderly people the number of peaks would be smaller as they are less mobile. For more active young people, there are much more peaks to process. Hence, it is not clear-cut anymore to decide where to run this component as the decision is tied to individual users and their life-style. Furthermore, this contextual dependency cannot be captured in the Pareto-fronts through profiling. In a third experiment, we tested the self-optimizing capabilities of the MDP on an individual with a sedentary lifestyle. The MDP picked up this behavior after on average 110 iterations, and finetuned the Pareto-curve with lower computation and network communication variability for deployment solutions D_4 and D_5, leading to an overall preference for the latter deployment.

6 Conclusions

In this paper, we presented our self-optimization approach for deploying IoT application components. The goal is to autonomously find the trade-offs between different component deployment configurations and their resource impact for distributed deployments on *Smart Objects*, *Smart Mobiles* and *Smart Servers*. Our approach is based on an offline exploration phase to collect relevant profiling information for optimization before actual deployment, and a runtime phase to autonomously adapt the deployment and configuration towards changing operational circumstances.

Our experiments have shown that the deployment and configuration decision (which part of the application is run on a sensor, mobile or a server in the cloud) is not always clear-cut, and that trade-offs are to be made w.r.t. application and QoS requirements. Our modular design philosophy for developing IoT applications helps to dynamically configure, compose and deploy these components depending on the QoS requirements of the applications. We have profiled and benchmarked these components on different deployment ends. This helped us to automatically find trade-offs for a distributed deployment of these components considering both the performance impact as well as the cost/benefit of any reconfiguration or change in component deployment.

As future work, we will explore the effects of more advanced learning and classification techniques and broaden our methodology to validate more complex deployment scenarios.

References

[1] BANDYOPADHYAY, D., AND SEN, J. Internet of things: Applications and challenges in technology and standardization. *Wireless Personal Communications 58* (2011), 49–69.

[2] BHATTI, Z., MINISKAR, N., PREUVENEERS, D., WUYTS, R., BERBERS, Y., AND CATTHOOR, F. Memory and communication driven spatio-temporal scheduling on mpsocs. In *Integrated Circuits and Systems Design (SBCCI), 2012 25th Symposium on* (30 2012-Sept. 2), pp. 1–6.

[3] BRANKE, J., MNIF, M., MULLER-SCHLOER, C., AND PROTHMANN, H. Organic computing-addressing complexity by controlled self-organization. In *Leveraging Applications of Formal Methods, Verification and Validation,*

2006. ISoLA 2006. Second International Symposium on (2006), pp. 185–191.

[4] CENSOR, Y. Pareto optimality in multiobjective problems. *Applied Mathematics and Optimization 4* (1977), 41–59.

[5] DEB, D., FUAD, M. M., AND OUDSHOORN, M. J. Achieving self-managed deployment in a distributed environment. *J. Comp. Methods in Sci. and Eng. 11*, 3, Supplement 1 (Aug. 2011), 115–125.

[6] DEB, K. Multi-objective optimization. In *Search Methodologies*, E. K. Burke and G. Kendall, Eds. Springer US, 2005, pp. 273–316.

[7] HU, Y., WONG, J., ISZLAI, G., AND LITOIU, M. Resource provisioning for cloud computing. In *Proceedings of the 2009 Conference of the Center for Advanced Studies on Collaborative Research* (Riverton, NJ, USA, 2009), CASCON '09, IBM Corp., pp. 101–111.

[8] HUEBSCHER, M. C., AND MCCANN, J. A. A survey of autonomic computing-degrees, models, and applications. *ACM Comput. Surv. 40*, 3 (Aug. 2008), 7:1–7:28.

[9] KEENEY, R. L., AND RAIFFA, H. *Decisions with Multiple Objectives: Preferences and Value Tradeoffs*. Cambridge University Press, 1993.

[10] KOEHLER, J., KOEHLER, J., GIBLIN, C., GIBLIN, C., GANTENBEIN, D., GANTENBEIN, D., HAUSER, R., AND HAUSER, R. On autonomic computing architectures.

[11] KOEHLER, M., AND BENKNER, S. Design of an adaptive framework for utility-based optimization of scientific applications in the cloud. In *Proceedings of the 2012 IEEE/ACM Fifth International Conference on Utility and Cloud Computing* (Washington, DC, USA, 2012), UCC '12, IEEE Computer Society, pp. 303–308.

[12] KWAPISZ, J. R., WEISS, G. M., AND MOORE, S. A. Activity recognition using cell phone accelerometers. *SIGKDD Explor. Newsl. 12*, 2 (Mar. 2011), 74–82.

[13] MARLER, R., AND ARORA, J. Survey of multi-objective optimization methods for engineering. *Structural and Multidisciplinary Optimization 26* (2004), 369–395.

[14] RAVI, N., DANDEKAR, N., MYSORE, P., AND LITTMAN, M. L. Activity recognition from accelerometer data. In *Proceedings of the 17th conference on Innovative applications of artificial intelligence - Volume 3* (2005), IAAI'05, AAAI Press, pp. 1541–1546.

[15] STEGHOFER, J.-P., DENZINGER, J., KASINGER, H., AND BAUER, B. Improving the efficiency of self-organizing emergent systems by an advisor. In *Engineering of Autonomic and Autonomous Systems (EASe), 2010 Seventh IEEE International Conference and Workshops on* (2010), pp. 63–72.

[16] SUNDMAEKER, H., GUILLEMIN, P., FRIESS, P., AND WOELFFLÉ, S. Vision and challenges for realising the internet of things. *Cluster of European Research Projects on the Internet of Things, European Commision* (2010).

[17] TESAURO, G. Online resource allocation using decompositional reinforcement learning. In *Proceedings of the 20th national conference on Artificial intelligence - Volume 2* (2005), AAAI'05, AAAI Press, pp. 886–891.

[18] TESAURO, G. Reinforcement learning in autonomic computing: A manifesto and case studies. *IEEE Internet Computing 11*, 1 (Jan. 2007), 22–30.

[19] TESAURO, G., AND KEPHART, J. O. Utility functions in autonomic systems. In *Proceedings of the First International Conference on Autonomic Computing* (Washington, DC, USA, 2004), ICAC '04, IEEE Computer Society, pp. 70–77.

[20] VENGEROV, D. A reinforcement learning approach to dynamic resource allocation. *Eng. Appl. Artif. Intell. 20*, 3 (Apr. 2007), 383–390.

[21] VERMESAN, O., FRIESS, P., GUILLEMIN, P., GUSMEROLI, S., SUNDMAEKER, H., BASSI, A., JUBERT, I., MAZURA, M., HARRISON, M., EISENHAUER10, M., ET AL. Internet of things strategic research roadmap. *Internet of Things: Global Technological and Societal Trends* (2009), 9.

[22] ZITZLER, E., LAUMANNS, M., AND THIELE, L. SPEA2: Improving the strength pareto evolutionary algorithm for multiobjective optimization. In *Evolutionary Methods for Design Optimization and Control with Applications to Industrial Problems* (Athens, Greece, 2001), K. C. Giannakoglou, D. T. Tsahalis, J. Périaux, K. D. Papailiou, and T. Fogarty, Eds., International Center for Numerical Methods in Engineering, pp. 95–100.