MIDDLEWARE-BASED SERVICES FOR VIRTUAL COOPERATIVE MOBILE PLATFORMS

A Thesis
Presented to
The Academic Faculty

by

Balasubramanian Seshasayee

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
College of Computing

Georgia Institute of Technology
August 2008

MIDDLEWARE-BASED SERVICES FOR VIRTUAL COOPERATIVE MOBILE PLATFORMS

Approved by:

Dr. Karsten Schwan, Advisor
College of Computing
Georgia Institute of Technology

Dr. Santosh Pande
College of Computing
Georgia Institute of Technology

Dr. Richard Fujimoto
College of Computing
Georgia Institute of Technology

Dr. George Riley
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Nitya Narasimhan
Motorola Labs

Date Approved: May 6, 2008

ACKNOWLEDGEMENTS

I owe much to my advisor Prof. Karsten Schwan for his constant guidance and yet providing me with the freedom to engage my creativity at the same time. I am deeply indebted for his words of encouragement when I needed them most, his patient handling of my shortcomings and the worry-free financial support and resources that he provided me throughout my Ph.D. studies. I thank my committee members: Prof. Richard Fujimoto, Dr. Nitya Narasimhan, Prof. Santosh Pande and Prof. George Riley for their feedback to improve my dissertation and for their many suggestions on my research directions. I have had the fortune of working with several smart people at Georgia Tech including Ripal, Himanshu, Ivan, Lakshmi, Pinar, Keith, Vibhore, Sanjay, to name a few. I have enjoyed their company, among many others', during my studies. I thank Dr. Greg Eisenhauer for sharing with me the EVPath and ECho software that he developed – which form an important portion of this thesis, and for patiently answering questions I had with them. I thank Keith O'Hara for sharing his robotics workload that I have used in this research. Thanks to Dr. Ada Gavrilovska and Dr. Matthew Wolf for helping me out with infrastructure issues. Above all, I thank my family for their patience, motivation and support throughout my studies.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Mobile computing devices like handhelds are becoming ubiquitous and so is computing embedded in cyber-physical systems like cameras, smart sensors, vehicles, and many others. Further, the computation and communication resources present in these settings are becoming increasingly powerful. The resulting, rich execution platforms are enabling increasingly complex applications and system uses. These trends enable richer execution platforms for running ever more complex distributed applications. This thesis explores these opportunities (i) for cooperative mobile platforms, where the combined resources of multiple computing devices and the sensors attached to them can be shared to better address certain application needs, and (ii) for distributed platforms where opportunities for cooperation are further strengthened by virtualization. The latter offers efficient abstractions for device sharing and application migration that enable applications to operate across dynamically changing and heterogeneous systems without their explicit involvement.

An important property of cooperative distributed platforms is that they jointly and cooperatively provide and maintain the collective resources needed by applications. Another property is that these platforms make decisions about the resources allocated to certain tasks in a decentralized fashion. In contrast to volunteer computing systems, however, cooperation implies the commitment of resources as well as the commitment to jointly managing them. The resulting technical challenges for the mobile environments on which this thesis is focused include coping with dynamic network topology, the runtime addition and removal of devices, and resource management issues that go beyond resource usage and scheduling to also include topics like energy consumption and battery drain.

Platform and resource virtualization can provide important benefits to cooperative mobile platforms, the key one being the ability to hide from operating systems and applications the complexities implied by collective resource usage. To realize this opportunity, this thesis

extends current techniques for device access and sharing in virtualized systems, particularly to improve their flexibility in terms of their ability to make the implementation choices needed for efficient service provision and realization in the mobile and embedded systems targeted by this work. Specifically, we use middleware-based approaches to flexibly extend device and service implementations across cooperative and virtualized mobile platforms. First, for cooperating platforms, application-specific overlay networks are constructed and managed in response to dynamics at the application level and in the underlying infrastructure. When virtualizing these platforms, these same middleware techniques are shown capable of providing uniform services to applications despite platform heterogeneity and dynamics. The approach is shown useful for sharing and remotely accessing devices and services, and for device emulation in mobile settings.

CHAPTER I

INTRODUCTION

1.1 Background

Rapid improvements in hardware technologies have resulted in a continuing proliferation of sensors and handheld devices connected by wireless networks. Microprocessors designed to target the mobile domain have grown in speed and complexity over the last two decades. For instance, as Figure 1 shows, the clock frequency of the ARM 32-bit microprocessor that dominates the mobile microprocessor industry, has been rapidly increasing over the years, presently touching 1GHz with the Cortex cores. The trend towards multi-core architectures in the desktop and enterprise computing platforms have also influenced the mobile processor market [2]. Developments in communication technologies have been equally rapid, with several competing wireless standards targeting various metrics such as price, performance, power, etc. Table 1 lists a few current and future technologies with throughputs as high as 4Gbps. These hardware developments are being accepted into mainstream products, fueled by high current and projected demands (Figure 2).

Positive trends in hardware have spurred developments in systems software as well, where mainstream operating systems like Windows, Linux [7], and MacOS [9], have been

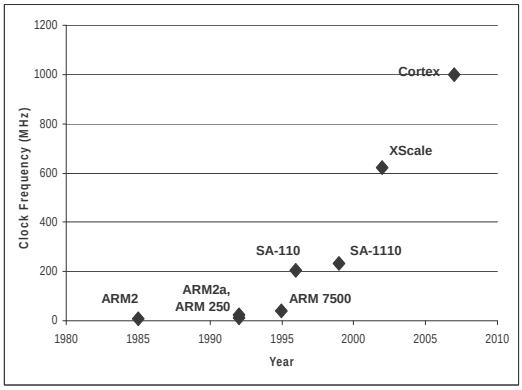


Figure 1: Trends in Clock Frequencies of ARM Processors

Table 1: Wireless Technologies and Data Rates

Wireless Technology	Maximum Data Rates
EDGE	120 Kbps
3GSM	300 Kbps
Bluetooth	3 Mbps
WiMAX	70 Mbps
Wi-Fi (802.11b)	11/54/100 Mbps
Wireless USB	110/480 Mbps
Ultra-wideband	110/500 Mbps
WirelessHD	4 Gbps

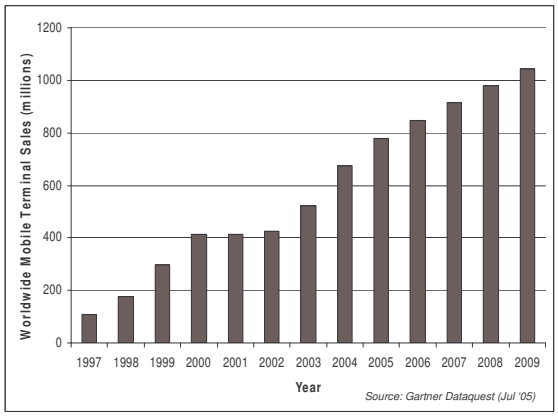


Figure 2: Current and Projected Global Sales of Mobile Phones

modified to target mobile platforms, in addition to operating systems like Symbian [11] and QNX [10] that are designed exclusively for such platforms. Further, the Java language [20] coupled with the Java Virtual Machine (JVM) was developed to provide a uniform interface for applications to run on mobile platforms, irrespective of the underlying hardware. Recently, this technology is being used as part of a toolkit [1] to speed up application development on such platforms. Finally, virtualization techniques that allow multiple operating systems to coexist on the same physical platform – currently extensively used in desktop and enterprise platforms – are also being redesigned for mobile settings, with Xen [71] and L4 [147] being two examples.

Advances in processor technologies allow resource-intensive applications to be successfully executed on ever faster chips. Improvements in communications enable inexpensive

sharing of device resources among each other, leading to more opportunities for cooperation and sharing. Platform-level cooperation among devices also helps improve resource sharing and manageability [89], where cooperation can range from simple interactions via remote procedure calls [29], to more complex ones, analogous to the use of SSI(Single System Image) operating systems such as Plan 9 [116] in a computing cluster. In mobile scenarios, MANETs (Mobile Ad hoc NETworks) provide opportunities for the cooperative execution of applications. Mobile devices in MANETs communicate with each other via ad hoc network connections requiring no networking infrastructure to operate. As a result, these are useful even when infrastructure is sparse, non-existent, or non-operational, as in disaster relief efforts. MANETs are also used in autonomous robotics [43] and vehicular networks [95]. In robotics, network interactions allow individual robots to share sensor data, behaviors learned over time, etc., and in vehicular networks, vehicles can benefit from traffic details in a remote area, in order to perform corrective actions in anticipation. Here, current systems suffer from limitations in communication capabilities, but improved systems like WiFi will further improve the opportunities derived from shared services or even low-level resources like the processor, memory capacity, for certain devices (e.g., sensors), translating to faster/better application behaviors. For instance, in well-networked systems, one robot could perform data processing on another's behalf, if the latter's battery level were low, or if it required faster response times.

Virtualization mechanisms introduce well-defined abstractions that simplify cooperation and sharing. Originally developed for enterprise computing, virtualization divides all physical resources into different and strongly isolated domains, termed VMs (Virtual Machines), and then mediates domains' access to the hardware via a thin software layer termed VMM (Virtual Machine Monitor). Since VMs are largely self-contained, VMMs also allow migration of a VM from one physical machine to another. This facilitates hardware maintenance, besides providing other benefits such as load balancing. Although developed for the enterprise domain, virtualization has also found widespread use in desktop environments, primarily for its isolation properties, and also because it allows legacy operating systems to

be run in their own VMs alongside others. More recently, there has been increased interest in extending virtualization to the mobile domain [71, 147]. Here, while security is one key focus of these efforts, designs also address other important issues. For instance, the support of VM migration provided on these platforms simplifies the migration of applications to appropriate neighboring physical devices depending on the context and on current power/performance requirements [121].

Virtualization can be used to help address challenges in the area of pervasive computing. Pervasive computing [154] refers to a mobile computing model where computers, in conjunction with sensor devices, surround the environment of a user and interact with him/her in a seamless fashion. In such an environment, the proliferation of devices poses problems, including discovering these devices, accessing them, and controlling their use, especially in a multi-user setting where sharing becomes necessary. Further complications are due to the need to support diverse applications and operating systems, each with their own device interaction mechanisms. As virtualization introduces well-defined abstractions to mediate access to physical devices, virtualization-based solutions help mediate such complexity.

1.2 Motivation

A basic challenge in MANET-based platforms is to deal with dynamics in the environment, which arise from nodes joining or leaving the network as well as from changes in connectivity due to mobility. Network-related services like routing have been developed to deal with such uncertainties (e.g., AODV [115]). For complex distributed applications to be run on cooperative networked nodes, however, additional support is needed for collectively managing the set of computing and device resources they require. Important problems in such settings include the following:

- managing energy resources to extend application lifetime: as mobile nodes depend on onboard power sources, careful management of energy consumption is necessary;
- balancing loads to meet end-to-end performance constraints: since load balancing involves distributing computation among multiple nodes, it is important to ensure

that applications' end-to-end requirements are not violated as a result of larger data exchange latencies;

- dealing with heterogeneity in terms of resource availability and device characteristics; and
- adaptation to dynamics arising from mobility.

Solutions to these problems must be scalable to large numbers of devices, offer low overhead, and be easy to adopt in practice.

On extending cooperative execution to virtual mobile platforms, we find that the features provided by existing virtualization solutions in sharing devices and computations among different machines are limited. While efficient methods exist to share resources among VMs in a single machine, only recently have research efforts started addressing the problems in managing resources among multiple VMs running in different machines [87, 150] and in sharing remote devices among VMs [84]. Issues to be addressed for enabling cooperative execution in virtual mobile systems include those below:

- providing methods to share devices among VMs running in different machines in a VM-transparent manner: implementing such methods at the platform level frees VMs from being aware of all the accessible devices, individually dealing with them and adapting to dynamics;
- enabling greater flexibility in sharing resources among remote VMs: the separation of a device from its use gives rise to challenges in matching the capabilities of the device to the using entity's requirements; since the platform manages remote devices on behalf of VMs, performing such capability matching becomes a difficult problem;
- allowing trading off the isolation properties provided by VMs for increased sharing and efficient execution: current virtualization technologies provide a tight separation between VMs; while useful for securing VMs from one another, this proves to be rigid for VMs desiring to cooperate with each other.

This dissertation develops and evaluates solutions to these problems using middleware-based services. Specifically, it adopts a service-oriented approach, where middleware is used to provide useful functionality to higher layers via a set of services. Higher layers can then utilize these services according to their needs. By reducing the dependencies on the middleware, such a design facilitates (i) easy adaptation of existing applications to the middleware, (ii) better security, by securing data exchange via the services, and (iii) flexibility in the use of these services. The approach leverages the fact that distributed middleware including CORBA [149], DCOM [32], EJB [127], etc., have been key enablers in cooperative platforms in the desktop and enterprise domains and that there are also effective middleware-based solutions for the mobile domain [16].

1.3 Terminology

Some of the terms used in this dissertation are defined below.

Middleware: Middleware is a generic term used to describe system software that is layered between an application and the operating system, and connects software components within the same machine or across machines. In the latter case, it is usually termed distributed middleware. Initially designed to allow inter-operation with legacy applications, later efforts have enhanced middleware systems to provide more complex functionalities to the supported applications. Interactions supported by middleware range from remote procedure calls to event-based messaging such as publish/subscribe.

Service: A service is a software abstraction that enables an implementation of a software functionality by one entity on behalf of another, via a common well-defined interface. The clean separation of an implementation from its use afforded by services have made them a popular option for separating levels of abstraction in software. Services may exist within a single machine (e.g., scheduling, memory management) or across machines (e.g., domain name service, web services).

Cooperative Platform: In the context of this dissertation, a cooperative platform is a distributed software framework that allows an executing application to access resources existing in any participating node in the framework and its underlying hardware. Since it allows cooperation at the platform level, its application interfaces are generic enough to support a variety of applications.

MANET: MANET, or Mobile Ad hoc NETwork, is a wireless network formed by mobile nodes where data transfers between two end nodes is carried out by other nodes belonging to the network, without relying on external infrastructure such as wireless access points.

Virtual Platform: A virtual platform, in the context of this dissertation, is a distributed software framework based on virtualization techniques that provides Virtual Machines (VMs) executing within the framework with access to resources belonging to its participating nodes, via a service-based interface.

1.4 Thesis Statement

Rapidly evolving hardware and software systems in the mobile domain make it possible to run powerful applications across multiple cooperating platforms, but this introduces new challenges arising from the need to manage resources under dynamics in a mobile environment.

The thesis of this dissertation is:

Next generation mobile systems enable the construction of cooperative and virtual platforms in ways that balance resource availability with applications' requirements. Flexibility in functionality and resource sharing on such platforms can be achieved using familiar service abstractions realized with existing middleware techniques.

The dissertation demonstrates how services constructed with state-of-the-art middleware techniques can be used to achieve:

- self-management of the collective resources of multiple mobile nodes, to continuously adapt an application running across such a cooperative platform to dynamics in its environment, in a decentralized fashion;

- providing a uniform interface to access and share devices in a cooperative virtual platform, using multimedia as example; and
- flexibility in trading off access and sharing of resources against isolation mechanisms in a virtual platform.

1.5 Organization

The dissertation is organized as follows. Chapters 2 and 3 focus on middleware in cooperative platforms, while Chapters 4 and 5 study the role of virtualization in cooperative platforms. Chapter 2 describes *Mobile Service Overlays (MSO)*, the distributed middleware framework used to support execution of an application over a cooperative mobile platform, and describes the services offered by this framework, while Chapter 3 demonstrates the use of these services to perform platform-level energy management. Chapter 4 discusses a device sharing framework in a cooperative virtual platform applied to multimedia devices, termed *VMedia*, and an extension of it termed *CustomCam* that allows functionality customization over the device interface. Chapter 5 describes *VServices*, a framework that performs service-level virtualization and demonstrates its use in mobile settings through examples. Chapter 6 discusses related work, and Chapter 7 concludes the dissertation and charts future directions.

CHAPTER II

MOBILE SERVICE OVERLAYS

Distributed applications running on Mobile Adhoc NETworks (MANETs) are being used in a wide range of domains, from autonomous robotics to emergency management. Examples include robots collaboratively undertaking a search and rescue mission [76], coordinated actions of geographically dispersed agents in an emergency rescue operation [99], distributed surveillance in the battlefield, distributed gaming, and offloading an application to surrounding entities in a ubiquitous computing environment [101], among others. Such complex applications require significant processing power from its underlying platform. If the underlying platform is distributed, as in the scenarios considered above, mapping portions of the application on to the various participating nodes in the network becomes an important issue to be addressed.

The dynamic nature of the MANET systems and applications considered in this work implies that the mapping problems formulated for prior work in static systems [31] have become dynamic mapping and reconfiguration problems. This requires solutions that are efficient in terms of their implied overheads and effective in terms of their ability to continually improve the Quality of Service metrics applied to distributed MANET applications. The Mobile Service Overlay (MSO) middleware makes the following contributions to addressing the dynamic resource management problems faced by MANET-based systems:

1. MSOs implement low-overhead computational overlays across cooperating distributed MANET devices;

2. MSOs provide efficient and scalable runtime abstractions, termed *chains*, for describing and then managing the overlays that run the computation graphs used by distributed applications;

3. MSOs provide efficient base support for online monitoring to assess the current resources used by and accessible to chains; and

4. MSOs permit developers to deploy alternative management policies based on different or multiple application-level metrics and SLOs.

2.1 Design Goals

MSO overlays are constructed and configured by the management layer of the MSO middleware. The design goals for MSO overlay management are the following:

- *Low overhead, low latency reconfiguration.* In a dynamic environment, the middleware, on detecting a change, must quickly arrive at an updated mapping, since the time available before the next change might be limited. In keeping with prior work on 'missed opportunities' in the real-time domain [126], this implies the need for efficient heuristic solution methods. Optimality is not the goal, because accurate system-wide resource information and precise knowledge about current application needs are not typically available for MANET systems.

- *Fault resilience.* Since cooperative solutions imply that participating nodes can arrive or leave dynamically, with similar effects caused by application mobility, MSO management must be resilient to change and failures. In contrast to full application-level fault tolerance, however, our goal is to have middleware stay "online" after faults occur. Applications can then realize their own fault tolerance methods on that basis.

- *Scalability through decentralization.* Centralized solutions will not scale in MANET environments. While respecting the dependence graphs of applications, the *chain* abstractions used by MSO middleware is the basis for localized configuration heuristics.

- *Localized reaction to changes.* The number of nodes that need to participate in reacting to dynamics (such as mobility, recovery after a fault or due to energy needs) must be kept to a minimum. Since decentralized solutions store the state of the system in a distributed fashion, it is important that the fewest number of nodes are involved in changes to the state.

In keeping with these goals, MSO middleware both (1) implements completely decentralized reconfiguration solutions and (2) enables remapping at various levels of granularity, ranging from a pair of nodes to the entire network. The MSO implementation consists of a set of services distributed over the network, so that identical instances run at each of the nodes. Each instance of MSO consists of a monitoring component, a decision-making component, and mechanisms for reconfiguration. Reconfiguration rules are used in the decision making process to decide which events obtained from the monitoring component can trigger the different levels of reconfigurations. The MSO middleware's design and implementation do not create additional dependencies across cooperating nodes. That is, each MSO node provides to the application an identical set of middleware services. The reason, of course, is that for MANET systems to attain failure resilience, each device must be able to operate independently of other devices.

Centralized solutions clearly do not work in the dynamics of a MANET environment. On the other hand, completely decentralized approaches allow scalability and fault resilience, but also replicate the state of the overlay network over several nodes, so that quick reconstruction of the overall state in the event of changes/failures becomes difficult. A cluster-based approach appears to be a compromise between these two extreme approaches, where the nodes are divided into clusters, and a special node, typically termed a "cluster head", is elected, to manage the state of its local nodes. This approach is shown to be efficient for routing in MANETs [136], since the communication among the cluster heads can be minimized through appropriate choice of the heads. However, the election of cluster heads itself requires computational effort, and further, the set of cluster heads might change over time. To avoid these costs, MSO lets the application flow graph automatically determine these special nodes. Instead of choosing special nodes based on several factors such as proximity of these nodes to other nodes, computation and communication capabilities, etc., as typically done in cluster head election, this election is implicitly performed using local metrics and the structure of the flow graph. The penalty paid for this benefit is that since the choice of cluster heads is automatic, it may not be optimal.

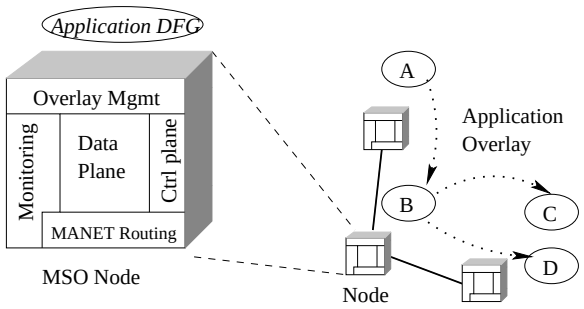


Figure 3: Overview of MSO's Architecture

2.2 MSO Architecture

2.2.1 Programming Model

MSO implements an event-based model of data exchange, as commonly used in pervasive applications like robotics and distributed simulation, as well as in enterprise computing [130] for complex event processing and in real-time systems [93]. The event-based model is known to provide a clean separation between different application services and their interactions. In this model, the application is represented as a directed flow graph, with each vertex signifying some processing and acting as a data source and/or sink. Data exchanges are represented as directed edges. Associated with each edge is a format that describes the data it transmits. MSO programs, therefore, consist of code modules running in vertices mapped to overlay nodes, receiving formatted inputs from and generating outputs to other vertices. The event-based programming model used by MSO is used by applications ranging from distributed sensor systems [83], to autonomous robotics [146], to multimedia [157], and even including select applications in the enterprise domain, as described in [93]. However, not all event based applications will be suited for environments targeted by MSO. For instance, windowing and similar component object model based applications commonly rely on event based interactions in order to maintain application neutrality. However, many such events are of a cascading nature (i.e., one event gives rise to several others), thus complicating the application structure and countervailing any benefits from distributing the application components. To run non-event based applications with MSO, they would have to be re-programmed. Typically, this would involve an analysis of their control and data flows and a subsequent decomposition of the application into modules communicating

via events [167, 45]. Such decomposition followed by a mapping to pervasive platforms would be impractical for applications in which program modules share substantial state and/or have tight dependencies.

2.2.2 The Chains Abstraction

MSO partitions the event-based application's flow graph into its constituent and independently manageable computational *chains*. Formally, a chain is a maximal set of sequential vertices and edges of the flow graph, with a single entry and a single exit. Events enter the first vertex of the chain, the *head*, sequentially pass through and are operated on at each node in the chain, and finally exit at the last node, the *tail*.

Chains also represent a smaller unit to use for runtime reconfiguration, thus reducing the graph mapping problem described earlier to the simpler chain mapping problem. In this fashion, chains compartmentalize management to be confined to more "local" portions of the application. The effects of compartmentalization are discussed further in Section 2.3.2. Chains also serve to capture desired end-to-end behaviors and node-node connectivity. Since an application's QoS can be formulated to depend on the QoS of each chain (e.g., consider end-to-end delay), each chain can be managed independently in order to guarantee such QoS properties (though not all QoS properties are composable in this manner [141].) Chains have also previously been used to enable computational offloading [107].

Algorithm 1 Chain formation

```
1: ∀ vertex v, unmark v
2: repeat
3:    Choose an unmarked vertex v and add it to an empty chain C giving it a unique id
4:    Set u ← v
5:    while ∃ unique unmarked pred(v) and v is its unique successor do
6:       C ← C ∪ pred(v)
7:       v ← pred(v)
8:       Mark v
9:    end while
10:   while ∃ unique unmarked succ(u) and u is its unique successor do
11:      C ← C ∪ succ(u)
12:      u ← succ(u)
13:      Mark u
14:   end while
15: until ∀v, v is marked
```

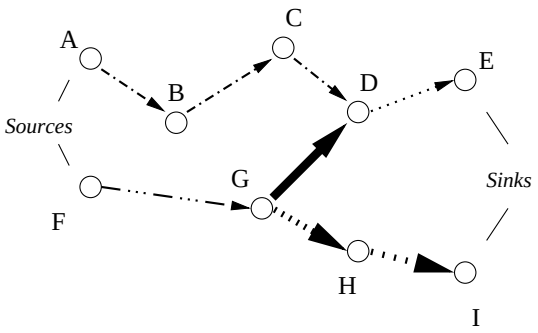


Figure 4: Partitioning the Graph into Chains

The algorithm used for chain formation (Algorithm 1) is straightforward. Intuitively, it selects some vertex and then creates a chain starting at this vertex, considering its successors and predecessors in the application's data flow graph. An example is shown in Figure 4, where the flow graph with source vertices A & F and sink vertices E & I, is partitioned into the list of chains (A-B-C-D, D-E, F-G, G-D, G-H-I). These chains have the property that there is one entry and one exit. A chain may have both a source and a sink, or only either of them, or even none at all. More importantly, the chain construction algorithm is of complexity $O(|V| + |E|)$ and can be run by each node in the overlay.

2.3 Services

MSO provides a few basic services to create and change the assignment (of the vertices of the flow graph to the mobile nodes). These services are then used by higher level management algorithms to effect policy decisions (Section 2.3.4). These services are described below:

2.3.1 Deployment

Application deployment involves mapping chains of the flow graph onto the underlying physical network, by assigning a node to each vertex in each chain, then constructing the overlay network based on this mapping, and finally, commencing data movement in the constructed overlay. Since more than one node participates in this effort, this is a distributed procedure initiated by the node to which the head of the chain is assigned. Deployment itself is subject to dependencies, where each chain is not deployed until all of the chains leading to it have been deployed. Hence, the deployment process is partially

parallel, dependent upon the average number of outputs of the flow graph vertices.

Algorithm 2 Deployment

1: **repeat**
2: If a chain has no dependences and the head of the chain is assigned to this node, then start the assignment by exploring the route taken by a packet to the sink node furthest away, collecting the capacities of all nodes along the route.
3: Allocate nodes to vertices of the chain in proportion to the fraction of the overall costs contributed by the vertices. Perform assignments of nodes to vertices.
4: Activate the node corresponding to the next chain by decreasing its dependence count and assigning the node to the head of its chain.
5: **until** all chains have been assigned to nodes.
6: Each node that has a chain head mapped to it constructs the actual overlay network along its chain.

In realistic systems, source and sink vertices can often be assigned only to certain nodes (e.g., those that possess sensors/actuators that produce or consume data). We therefore, assume the source and sink vertices to be preassigned to particular nodes. If this is not the case, a distributed process can be used by which nodes possessing the capabilities required by the source/sink vertices are chosen according to their abilities and assigned to nodes before deployment.

As indicated in Algorithm 2, route exploration is a key step of the deployment process. MSO uses probing for this purpose, by sending a probe packet to a sink node furthest away, and obtaining the capacities of each node along the route taken by the probe packet. These capacities constitute metrics like processing capability, battery lifetimes, etc. To better illustrate the route exploration procedure, consider a simple example, where a flow graph consisting of three chains A-B-C-D, E-D, and D-F is mapped onto a network with four nodes, with the connectivity between them represented as N_1-N_2-N_3-N_4. Suppose that A is preassigned to N_1 and F to N_4. Now, while assigning the first chain, the route taken is obtained (N_1-N_2-N_3-N_4), and each node's capacities are queried in the process. Then, chain A-B-C-D is assigned a fraction of nodes N_{abcd} from among N_1,N_2,N_3,N_4 such that $\frac{cost(N_{abcd})}{\Sigma(Nx)} = \frac{cost(A-B-C-D)}{cost(A-B-C-D)+cost(D-F)}$. Now assume that this leads N_{abcd} to be N_1-N_2-N_3. Then, B, C, and D are mapped among N_1-N_2-N_3 (A is already mapped to N_1). Note that deployment is linear on the number of vertices and network nodes, and uses a greedy method to assign vertices to nodes. While it is therefore not optimal, the MSO middleware

continuously seeks to improve local optimality metrics through its intra-chain remapping services discussed in the following section.

2.3.2 Reconfiguration Services

Reconfiguration involves remapping portions of the overlay network, to create a different assignment between vertices in the overlay and underlying machines. The primary goal of MSO is efficient reconfiguration, and for this purpose, it provides capabilities to perform remapping at various granularities, as detailed below. Some assumptions are made by our current mapping and remapping methods: (1) the flow graph is a directed acyclic graph, and it remains static throughout the lifetime of the application, (2) node discovery, naming, and message routing are performed at a lower layer independent of the middleware, (3) all faults are fail-stop, and (4) the "source" and the "sink" modules of the flow graph are pre-assigned to specific nodes. We observe that these assumptions serve to simplify the implementation of MSO but are not the limitations of the approach itself. For instance these assumptions, may respectively, be relaxed as follows: for (1) any changes to the application flow graph are propagated to all participants, each of which update their own flow graphs; the deployment of new chains that have not been deployed yet, are triggered by their respective chain heads, and removal of old chains is performed similarly, for (2) other research such as [41, 67, 115] can be leveraged, for (3) other types of faults may be handled using replication [63], and finally, for (4) discovery of source and sink modules using techniques such as [41] may be used. In our current implementation, the state within a node is not transferred during a reconfiguration. Adding this functionality is straightforward when the nodes involved in the reconfiguration remain 'alive'. In cases when the node dies or loses connectivity (due to mobility, for instance), more involved mechanisms such as the use of a hot standby prove useful.

2.3.2.1 Intra-chain Remapping

Intra-chain remapping may be used continuously throughout the application's lifetime. It is triggered by monitoring events reporting changes in local resource availability and/or in application requirements or resource usage. An intra-chain remapping constitutes relocating

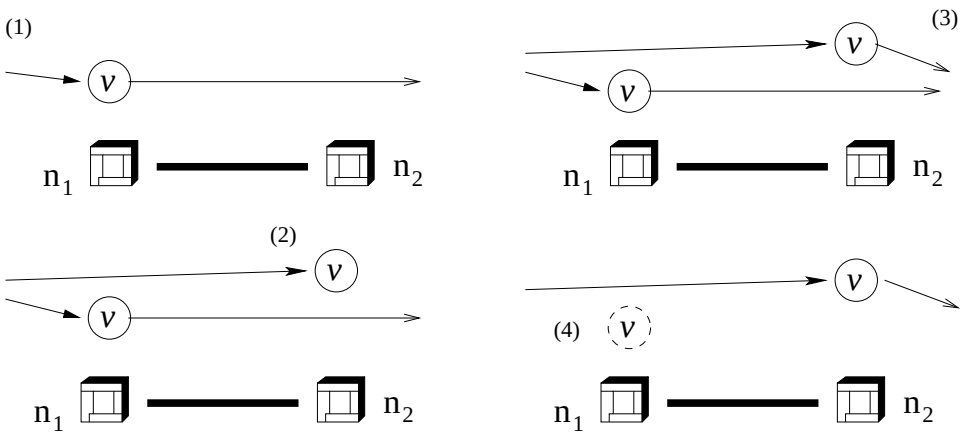


Figure 5: Intra-chain Remapping Example

a single vertex from one node to another. The steps involved in a typical relocation, where a vertex v is moved from n_1 to n_2, as shown in Figure 5, involve (1) suspending event sends at each of v's predecessor vertices, (2) creating a new v vertex at $n2$ and (2) connecting it to the same destination as the previous v, followed by (3) linking the output from each of v's predecessors to the new v, and then (4) freeing up the old v. Note that these remapping strategies affect only the nodes in the vicinity of the change (i.e., local nodes), and other nodes in the network are neither involved in nor aware of this remapping. This process has low overhead and can be run fairly frequently.

Intra-chain remapping is useful in obtaining a local optimum. That is, based on the metric we seek to optimize, each node housing a vertex evaluates the cost of relocating it, against making no changes. The costs of relocating state associated with the vertices are to be included in estimating total costs. Each vertex can independently (but sequentially) perform this check and carry out the remapping.

2.3.2.2 Chain Remapping

In contrast to intra-chain remapping involving only two nodes, chain remapping affects all of the nodes to which a chain is mapped. The "head" and "tail" of the chain remain unchanged. The operation is performed in three stages. First, the chains chosen for remapping are reassigned to the nodes after a route exploration, ignoring the existing mapping. Next, the edges that lead to the older instances of the chains are rerouted to the just-assigned ones.

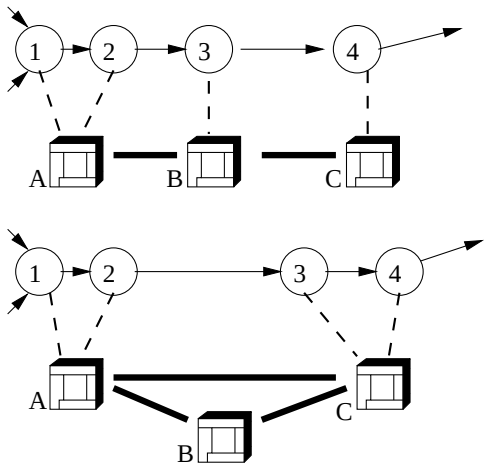


Figure 6: Chain Remapping Example

Finally, the overlay network corresponding to the older instances of the chains is freed. This procedure is illustrated in Figure 6, where the shorter route from A to C achieved due to node mobility results in remapping the chain along the new route, with the result that vertex 3 is now mapped to node C, from node B. Chain remapping is useful in the event of a change in route between the nodes housing the head and tail of the chain. Under these circumstances, only the affected chain needs to be remapped, leaving other chains unchanged.

2.3.2.3 Multi-Chain Remapping

Multi-chain remapping is carried out to effect changes that involve more than a single chain, and it involves performing the mapping operations described in Section 2.3.1, over a set of the chains. Synchronization between the nodes during the remapping process is enforced by the dependence relationship between chains, and it is achieved by remapping a chain only after all chains depending on have been remapped. Note that the procedure is identical to the mapping methods described earlier, except that in general, only a subset of the chains are remapped. Global remapping is useful when node failures or movements affect the head/tail of a chain, thus affecting multiple chains. It can also be used to remap all the chains, i.e., entire flowgraph. Due to the high cost and involvement of a number of nodes, this is used relatively less frequently, in comparison to the other remapping techniques discussed before.

As noted previously, these remapping schemes also incur costs proportional to their granularity, and it is important to consider these costs, before deciding to apply them. During unavoidable cases such as recovering from a node failure, one may not have an option in the remapping type. However, for remapping actions taken to meet an objective, such as energy efficiency, this becomes important. This point is discussed in more detail in Section 3.1.2.

2.3.3 Monitoring Services

Each MSO node runs a separate monitoring thread that is used to maintain metrics like a measure of the amount of computation carried out, the amount of data transferred from/to the node and the expected lifetime of the battery. Similar metrics are also maintained at the vertex level. In addition, monitoring also periodically checks for the liveness of its neighbors (as determined by the routing layer) and for changes in the routing layer. The metrics monitored are available for use by the higher layers for remapping decisions. This part of the subsystem requires access to the routing layer to detect any changes, but does not depend on the type of routing protocol used. Monitored metrics are also available for sharing, as each monitor exposes its data to other nodes through SOAP calls. In addition to per-node metrics, per chain and application level metrics may also be maintained. All the node-level and vertex-level metrics discussed here have been implemented and are made available to other nodes via SOAP calls.

Monitoring services are typically designed to be lightweight, in order to allow them to be invoked repeatedly. For instance, measuring battery levels, CPU frequency and utilization, are very inexpensive, as these are based on statistics already made available by the operating system (e.g., via `/proc` in Linux). On the other hand, monitoring remote metrics also add a SOAP call to the basic costs, and are more expensive in comparison. For this reason, these are performed less frequently. As an example, in experiments conducted to evaluate MSO's energy-aware reallocation (Section 3.1.2), monitoring neighboring nodes' energy levels are performed once every few minutes. Monitoring of protocols can lead to several insights on the current state of the system that cannot be captured otherwise, however, this may entail

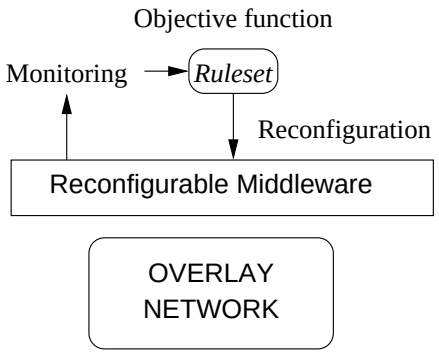


Figure 7: Generic Management in MSO

heavy costs for deconstructing the packets and/or prior knowledge of the protocol itself. Although MSO doesn't rely on protocol monitoring, support for this can be added with modest effort.

2.3.4 Management Services

MANETs undergo dynamics in connectivity among the nodes due to mobility, possible changes in the application's resource requirements as well as characteristics of the nodes themselves, such as energy levels, utilization, etc. Depending on the severity of the change, a remapping can be performed at the appropriate level of granularity, as a counter-measure. The decision as to what type of remapping is to be carried out, and when, is taken based both on continuous monitoring of the environment and on the formulated management rules.

As shown in Figure 7, a generic management framework in MSO involves (i) monitoring for specific changes in observed metrics, and (ii) triggering reconfiguration mechanisms at the appropriate granularity to counter the change, based on a predefined ruleset. As MSO is decentralized, any node that observes a change can trigger a reconfiguration according to predefined rules tailored towards specific objectives. Further, as this is a generic procedure, it can be applied to different management goals like load balancing, mobility & fault resilience, latency minimization, etc.

Management rules describe the action that needs to be taken in response to monitored events. For example, as discussed in Chapter 3, energy management relies on monitored

events such as the battery levels, as well as CPU utilization, frequency and slack periods of the application. In response to these events and predefined rules, actions such as performing offloading, using and propagating slack, etc. are carried out.

2.4 Evaluation

Mobile Service Overlays has been implemented in C/C++ using the EVPath [4] toolkit for creating and managing overlay networks. EVPath uses processing entities called *stones* that perform processing and routing actions. Associated with each stone is a queue, and stones are connected via links that support a variety of transport protocols. EVPath also provides a SOAP interface using gSOAP [49] to enable remote management of stones. EVPath uses PBIO [113] binary format for data exchange. PBIO supports several data formats using receiver-side data conversion, and permits native formats when both the sender and the receiver share the same platform characteristics. Kernel AODV [81], an AODV implementation for Linux 2.4, is used to perform routing. MobiEmu [165] is used to emulate the wireless network topology. MobiEmu uses the kernel netfilter to filter packets. The hardware platform used for the experiments is a *Sitsang* handheld computer prototype platform (Figure 8) described in Section 3.2.

2.4.1 Microbenchmarks

We conduct a set of experiments to determine the costs for the basic deployment and reconfiguration services provided by MSO. This is evaluated using a single chain deployed over a set of nodes in a network with a linear topology. The overheads resulting from deployment, and chain remapping are presented for different chain lengths (and consequently, different network sizes), with the costs of performing a null SOAP RPC call presented alongside for comparison. The results are shown in Figures 9 and 10, respectively.

With the null SOAP calls, the cost of each call increases linearly with the number of nodes in the network. Due to the topology, as the number of nodes increases, so does the end-to-end hopcount and latencies, thus increasing the cost of a SOAP call proportionally.

With the chain deployment and remapping costs, however, the increasing hop count and the increasing chain lengths compound to show a higher increase in the costs. In the

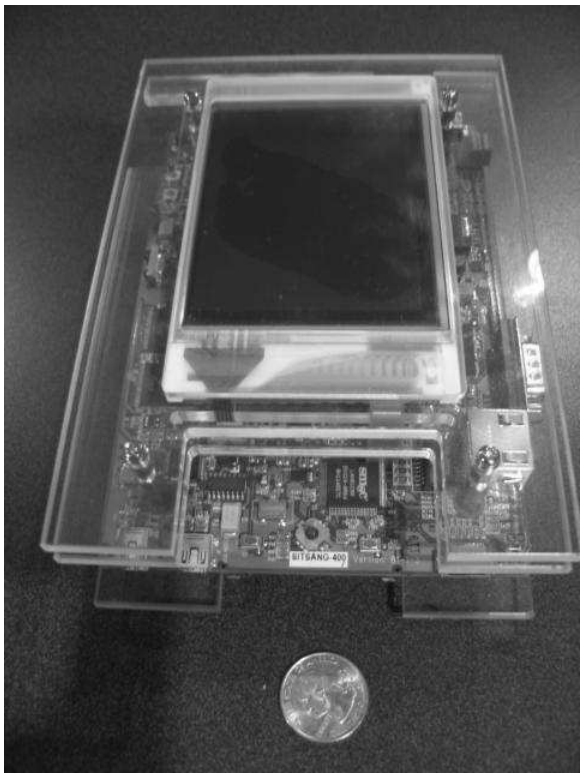


Figure 8: The *Sitsang* Handheld Prototype

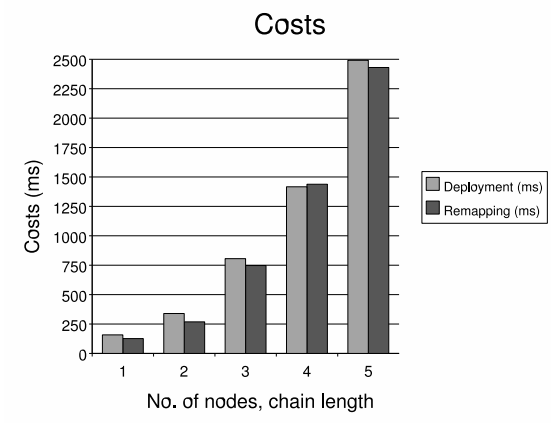


Figure 9: Chain Deployment, Remapping Costs

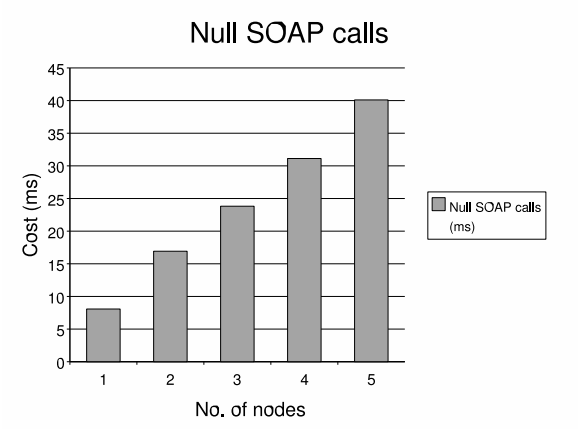


Figure 10: Null SOAP RPC Costs

Table 2: Overheads for Intra-Chain Remapping

Operation	Cost (μs)
Freeze predecessor	902
Create new vertex	57064
Connect the new vertex	267025
Free old vertex	33958
Unfreeze, update capacities	34094
Total	414,229

case of deployment, the main costs involve (i) determining the capacities of all nodes in the path and performing a suitable allocation of vertices of the chain to nodes in the path, and (ii) invoking EVPath calls to create stones according to the allocation and creating the links between them. While step (i) shows a linear increase with increasing path length (the computational cost of assigning vertices to the route obtained is negligible), step (ii) requires creation of more vertices (with increasing chain lengths), at greater hop counts (with increasing number of nodes) from the source node. The cost of chain remapping is comparable to that of deployment. In addition to the steps involved in deployment, in remapping, the chain head also frees the old chain and updates the capacities of each node before proceeding with the deployment. These steps incur additional overheads. However, during the initial deployment phase, the routes are computed on-demand, during the route exploration step (since AODV is a reactive routing protocol). This cost is absent during remapping, as the routes are already computed.

Next, the costs involved in performing an intra-chain remapping are studied. Using the

same experimental setup as the previous experiment, we randomly move a vertex from one node to its neighbor among nodes in the five-node linear network. The various costs involved are shown in Table 2. The overall cost is about $400ms$, on an average. Of this, almost half the cost is due to making edges between the predecessor of the vertex and the new vertex, and between the new vertex and its successor. This translates to multiple SOAP calls to at least three different nodes, as well as dependencies among the SOAP calls themselves (i.e., since the calls are blocking, this leads to a cascade of dependent calls). In comparison, other operations involve calls only to one or two different nodes with no dependencies.

2.4.2 Sample Application

The next experiment is conducted on a netlab testbed [106], consisting of four nodes. The CPUs are run at 350MHz, in order to mimic embedded computers. We construct a robotics application to mimic a scenario with four robots proceeding in a convoy, in a search-and-rescue mission. The application consists of a navigation pipeline, where laser and odometry data from sensors are used to locate the current position of the robots in a map, and to plan a path through it. It also consists of a target recognition pipeline, where images from visible light and infrared cameras are processed and Bayesian inference used to estimate the chances of finding a target in an area. The results from both these pipelines are then stored for later analysis. Of the four robots that cooperatively run this application, the leader robot is assumed to possess the sensors, and the trailing robot, the storage. Clearly, running all of the above tasks in the leader robot due to the proximity of processing to data, and communicating the results to the trailing robot, will drain the former's batteries quickly. Hence, MSO can be applied to offload the computations to the intermediate robots.

We model this scenario using four nodes lying in a straight line, with each node connected only to its physical neighbors. The application makes use of the localization and navigation modules from CMU's CARMEN [36] robotics software suite, and CMVision for image analysis. Other parts of the application are developed in-house. The data for laser and odometry sensors were taken from a sample simulation run of CARMEN. The data sizes from these sensors are 967 bytes and 38 bytes, respectively, and are sent at the rate

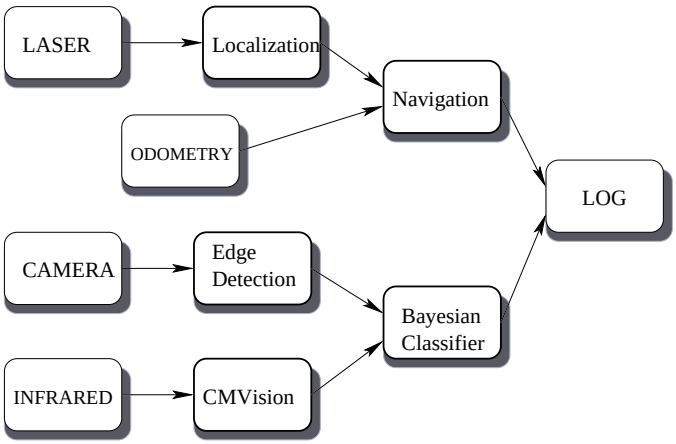


Figure 11: DFG of Robotics Application

of 5 events/sec. The images used in place of the cameras are scaled down to roughly 3KB in size and sent at 0.5 events/sec. The resulting flow graph is shown in Figure 11. All the computational vertices of the flow graph are assigned the same cost, and the capacities of the nodes are allocated such that each node in the network is assigned at least one vertex.

The overheads of mapping and remapping the flowgraph onto the four node network are measured over several iterations (Table 3). The cost of mapping the flowgraph is around 1.4 seconds. A large portion of this cost is due to route exploration by the nodes along the linear path. However, since data transfer has not commenced, the nodes are lightly loaded and overhead is low. Table 3 shows the costs of performing remapping at various granularities – remapping the laser-navigation chain alone, the entire navigation pipeline, and finally, the entire flowgraph, respectively (Figure 11). These costs are significantly higher, arising due to the data volume passing through the nodes. The high variations in their values (σ in the table) is also due to this. This causes a slowdown in the SOAP-based control messages, which form a dominant portion of the remapping processes. Indeed, when run at 2.8GHz, these costs went down by as much as 10x. To overcome this effect, a more efficient cross-platform RPC protocol could replace SOAP. For instance, our own previous work [133] has studied the overheads of XML in SOAP and proposes alternatives. Other optimizations like queueing data during remapping, asynchronous remapping, etc. may also be applied, in order to minimize the interference of data traffic on control.

Table 3: Overheads for the Robotics Application

Operation	Cost (ms)	σ(ms)
Mapping	1,390.0	77.7
Chain remapping	2,759.1	217.0
Multi-Chain remapping (par)	2,943.1	265.8
Multi-Chain remapping (whole)	3,327.2	335.2

2.5 Discussion

The MSO middleware follows a modular design in permitting distributed applications to be run on MANETs, consisting of monitoring and reconfiguration services, rooted in the chains abstraction. This design allows other services to be added to this list, as well as permitting high level decision making algorithms to offer specific functionalities. As a result, MSO can be implemented in Java or using virtualization, without departing from its basic design. However, the data-flow nature of MSO-supported applications is essential, in order for the middleware to be aware of modular dependencies. Applications already written to be compatible with component-based middleware implementations can be, in principle, adapted to MSO, as these dependencies are already made explicit. Finally, the current implementation of MSO relies completely on simple request-response based interactions. It uses SOAP for control messages, and as mentioned previously (Section 2.4), may be replaced with other cross-platform RPC mechanisms that incur lower overheads. The rationale behind the request-response interaction style arises from the peer-based environments that MSO is designed to operate in. Consequently, more complex models of interaction such as multicast or other group communication methods will add maintenance overheads.

The absence of any organization (such as centralized, hierarchical, etc.), although designed to address node dynamics, also serves to improve MSO's scalability. MSO uses the AODV protocol for routing, which has been shown to be scalable [115]. Algorithm 2 carries out the mapping of graph vertices to nodes by mapping the individual chains, whose mapping cost is dependent on the route taken by a packet from the source node to the sink. Similarly, the costs of remapping depend on the number of nodes involved in the operation. Additionally, monitoring services are carried out independently in each node. The result of these design choices makes MSO highly scalable.

MSO supports a computing model where all the nodes involved in running the application explicitly change their functionality within the cooperative effort – i.e., by relocating vertices of the application's flow graph from one node to another. This model can be modified to support computational offloading, whereby such functionality relocation is carried out without prior sharing of codes. In such cases, the offloaded node needs to be supplied with the code in a form that it can execute, in an isolated manner. Other features such as resource accounting, fault management, Service Level Agreement (SLA) negotiation, etc. also need to be supported in order to run applications in a delegative fashion. MSO's monitoring and reconfiguration services can be used to implement the first two features respectively, whereas SLA negotiation and mechanisms for code exchange need to be added to the existing implementation of MSO, in order to extend it to support such applications.

CHAPTER III

ENERGY MANAGEMENT WITH MSO

This chapter discusses *cooperative energy management strategies* developed for MANET applications, to enhance energy profiles, (i) by decreasing energy consumption to the extent permitted by current application performance constraints, and (ii) by extending overall system and application lifetime, via migration of application services that are critical to the application away from energy constrained nodes. Specifically, concerning (i), for each single platform, we reduce its energy consumption by using common techniques for energy management, which include dynamic voltage and frequency scaling (DVFS). The resulting degradation in application execution can be reduced by utilizing memory-bound phases for such scaling [72], or, in real-time environments where there is a notion of slack, by increasing execution times (thereby reducing energy needs) only to the extent permitted by application deadlines [22]. Similar energy-performance tradeoffs are available for other devices like memory, peripherals (network and disk interfaces), display, etc. Concerning (ii), we use computational offloading, whereby portions of the mobile workload are dynamically offloaded to nodes with better energy resources [117]. The former set of techniques have a direct effect on energy savings at distinct nodes, whereas the latter helps in longevity by sharing energy resources amongst multiple participants.

The reconfiguration and monitoring services discussed in the previous chapters are used to develop decentralized management protocols that (i) dynamically distribute and re-distribute application components among participating nodes, considering the overlay routes that satisfy the application's latency requirements while at the same time, determining the most energy-efficient allocations, (ii) recover unused portions of resources in an overprovisioned system with little or no impact on application performance, and (iii) use de-centralized online monitoring and reconfiguration to locally, and thereby, with low

delay and overhead, respond to dynamic changes in application requirements and environment conditions. The management algorithms being used, specifically the algorithm for dynamic resource reclamation, are experimentally demonstrated to track optimality, with low overhead. MSOs using this algorithm – *energy-aware MSOs* – offer notable benefits. On a wireless, multi-hop ad-hoc network of handheld computing platforms, for instance, an energy-aware MSO extends system lifetime up to 10% for a five-node network.

3.1 Techniques for Energy Management

Energy management in a dynamic environment is a continuous process, requiring an energy-aware assignment, followed by online monitoring to trigger actions that shift the system towards optimality. MSO provides three techniques for energy management, viz., energy-aware allocation, reallocation, and dynamic resource reclaiming.

3.1.1 Energy Aware Allocation

The chain deployment procedure (detailed in [131]) is modified to address energy management concerns, using the following techniques: (i) modifying route exploration to include *Ad-hoc Route Neighborhoods* and (ii) using the *Global Lifetime Sustainability* heuristic to determine the best assignment of vertices to nodes from among the routes in the neighborhood.

Ad-Hoc Route Neighborhoods: The allocation algorithm in MSO uses the route explored from the source node(s) to the sink(s) to perform the assignment over each chain. However, the route thus found is entirely dependent on the ad-hoc routing protocol employed in the underlying layer. While protocols like AODV, DSR, etc. typically use the shortest route, recent research has explored a variety of techniques for power management at the network routing layer, including probabilistic routing [134], multipath routing, and using multiple radios. Hence, prior to allocation, no assumptions can be made by MSO about the underlying protocol's behavior. As a result, MSO explores all possible routes that can be taken from the source to the destination (bounded by a maximum hopcount), and chooses the "best" route from among those for the assignment. The set of all the routes

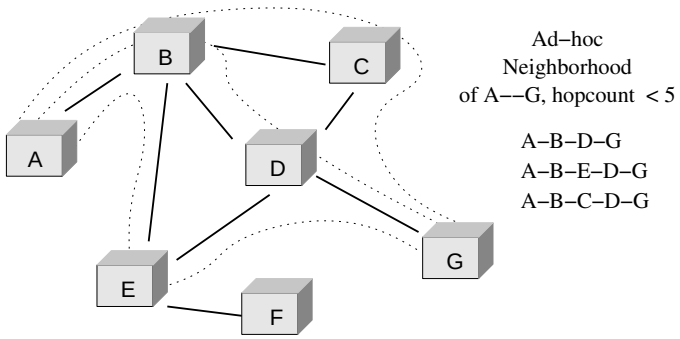


Figure 12: Ad-hoc Route Neighborhood

discovered in this manner is termed the *ad-hoc route neighborhood* of the chain. The end-to-end latency requirements of the chain limits the maximum hop count for the route, and consequently, the length of each route in the ad-hoc neighborhood, and the number of routes obtained. Figure 12 illustrates this concept, enumerating the ad-hoc route neighborhood of routes from node A to G in the topology, with hopcount strictly less than 5.

We rely on a distributed protocol to find the route neighborhood of a chain, given constraints that include maximum latency and maximum hopcount. The protocol is similar to that used by AODV for route discovery [115], with two differences: (i) the destination node stores all the routes it obtains, and (ii) after the maximum latency period, the source node queries the destination to directly obtain all the routes stored. This protocol enumerates all of the routes satisfying the latency constraints. Since no state is saved in any nodes (except at the destination), its overhead is low. Further, each packet not reaching the destination is eventually dropped, as the monotonically increasing hopcount reaches the upper bound.

Global Lifetime Sustainability (GLS) Heuristic: The placement of software components during allocation and reallocation can directly affect the lifetime of a MANET scenario. In the simplest case, with two nodes and a single component, energy can be depleted fairly and system lifetime is maximized by migrating the workload to the participant with larger battery capacity whenever a reallocation is triggered. When there are multiple components and multiple nodes, all with varying requirements and energy levels, optimal decisions cannot be made within reasonable complexity constraints. Instead, we use a heuristic to determine where to place various components. In particular, we define

a GLS metric for a set of candidate nodes and energy levels and utilize a heuristic that attempts to maximize this metric and/or minimize potential decreases in metric values. Specifically, for this work, we define the GLS metric to be the product of the remaining energy levels on nodes under consideration when deciding where to place a software component. This approach works well for the scenarios addressed here since nodes are treated equally and are homogeneous. We note that the GLS metric can also be effective when used in more general settings, to express variations among nodes in heterogeneous environments (since power consumption profiles vary accordingly) and when tuning allocation policies at runtime.

Chain assignment decisions in MSO use the GLS heuristic and ad-hoc neighborhoods. Intuitively, from among all possible routes discovered (ad-hoc route neighborhood) for each chain, (1) a greedy assignment is determined based on resource availability and the GLS heuristic, and (2) a cost is associated with the route and allocation scheme. The route with the smallest cost is then chosen for the chain. In particular, we define the cost as the projected difference in the GLS metric based upon the allocation over some period of time. Additional detail about the algorithm appears in [131]. The GLS heuristic arises from earlier research efforts on energy-aware assignment of workloads to data centers with heterogeneous platforms [104].

3.1.2 Energy-Aware Reallocation

Energy-Aware reallocation consists of moving application components in response to changing conditions. It utilizes the monitoring, management, and reconfiguration services provided by MSO (see Algorithm 3). Here, local variations in node lifetimes are overcome by relocating vertices to neighboring nodes with higher lifetimes. Global variations are addressed by remapping entire chains, but this is performed less frequently due to its higher costs.

Algorithm 3 does not factor the cost of data transfers into and out of a node (for a processing vertex housed here). Hence it is suitable only if these costs are negligible, and the energy drain is dominated by compute-bound processes. However, with data intensive

Algorithm 3 Energy-Aware Reallocation

```
1: Each node n, periodically queries the expected lifetime of all of its neighbors
2: Select node n' s.t. lifetime(n') is the largest among all neighbors
3: if lifetime(n') − lifetime(n) > threshold then
4:    while ∃ v' in n not considered for relocation do
5:       Select vertex v s.t. Cost(v) > Cost(v'), ∀v' in n not considered for relocation
6:       if v's relocation to n' does not affect event latencies along the chain then
7:          Relocate v to n'
8:          Break out of while loop
9:       end if
10:   end while
11: end if
12: During long periods of idleness, ∀v housed at n : v is a chain head, check for any changes
    in the ad-hoc neighborhood, and remap the chain if a more optimal neighborhood is
    found.
```

applications these costs must also be considered. For example, consider a local vertex that gets its input from and provides output to other local vertices. Further, assume that the vertex corresponds to data-intensive processing. Relocating this vertex to a neighboring node is clearly not energy-efficient, because more energy will be expended in the data transfers, rather than the processing. To allow these factors to be considered, the following metrics are monitored for each vertex in the flow graph:

1. Event input rate (in bytes per second).

2. Event output rate (in bytes per second).

3. Processing cost (CPU utilization, time to process an event).

These metrics are averaged over a period of time, before being used in the decision making. Further, the energy associated with data transfers, and CPU utilization are also considered. Using this method, we can express the three quantities in the same units, (watts, for example). Let us denote these three quantities by i, o, and c, respectively. Note that these metrics are platform dependent. For instance, in the *Sitsang* platform, the data transfer rates are also dominated by the CPU utilization (due to packet processing). It is not important that these values are highly accurate, for the simple fact that these are used only for decision making in vertex relocation, and hence estimates are sufficient. For the Sitsang,

the quantities i and o roughly equate to 300mW at 4Mbps (the peak bandwidth achieved). The values for c can be obtained using the methodology outlined in Section 3.2.

After these metrics are estimated, the decision rules are formulated as follows:

Algorithm 4 Vertex Relocation

```
1: if Input of the vertex is outside then
2:    if Output of the vertex is outside then
3:       Always perform vertex relocation
4:    else
5:       Perform relocation if i + c > o { Node pays for input and computation}
6:    end if
7: else
8:    if Output of the vertex is outside then
9:       Perform relocation if o + c > i { Node pays for output and computation}
10:   else
11:      Perform relocation if c > i + o { Node pays only for computation}
12:   end if
13: end if
```

The rationale for these decisions result from energy cost-benefit analyses of retaining the vertex in the old node versus relocating it to the neighbor. Since each node initiates vertex relocations only on those vertices that it hosts, it computes the tradeoffs in maintaining the vertex at the same location versus offloading it to another node, before making the decision.

3.1.3 Workload-Aware Dynamic Resource Reclaiming

To conserve energy in overprovisioned nodes, we design a distributed protocol that explores energy-performance tradeoffs in a distributed system through resource reclaiming – i.e., recovering any resource from the system to the extent that it does not affect the performance and hence, the quality of the application. Such resources include peripheral interfaces like storage (via sleep modes), memory (via switching off banks), and CPU (dynamic voltage/frequency scaling). In this work, we demonstrate this approach with CPU-based techniques.

Each node attempts to reclaim as many of the resources as possible to minimize energy consumption, then distributes the remaining opportunities to other nodes. Event sources in MSO associate a deadline with each event, sending it along with the event itself. Where this is not available/known, the event inter-arrival period is used. When the processed event

reaches the sink, its slack is computed, i.e., the time difference between the deadline of the event processing and the actual time of completion. A positive slack value serves as a measure of overprovisioning, in that unnecessary effort was expended in processing the event. This presents an opportunity to 'reclaim' the slack by scaling down the voltage/frequency in some/all of the CPUs involved in event processing, thereby reducing energy consumption. An implicit assumption made here is that the nodes' clocks are synchronized.

Slack reclaiming starts at the sink node, since only it can compute the slack value. Starting from this node, each node, on obtaining the slack values from all its downstream nodes (a node n_1 is downstream to n_2 if $\exists v_1, v_2 : v_1$ is assigned to n_1, v_2 to n_2, and $\exists$ a directed edge from v_2 to v_1), attempts to scale down its own CPU frequency/voltage so as to maximize energy savings. Next, it computes the slack available to each of its upstream nodes and sends them these values. Starting at the sink node(s), the algorithm thus propagates towards the source(s).

Algorithm 5 Slack reclamation

1: **repeat**
2: **if** $\exists v : \forall v'$ *and* $v \rightarrow v'$, v has received $slack(v')$, **then**
3: **for all** $f : f$ is a CPU frequency **do**
4: Set $slack_f(v) = min_{v'}\{slack_f(v')\} - t_f(v)$, { where $t_f(v)$ is the estimated execution time of the computation of v at frequency f}
5: $\forall v_p : v_p \rightarrow v$ *and* v_p is in the same node as v, send $slack(v)$ to v_p
6: **end for**
7: **end if**
8: **until** all the vertices in the node have been considered {Slacks for each frequency have been computed}
9: **for all** $v : v'' \rightarrow v$, and v'' is not in this node, **do**
10: $f_{choose}(v) \leftarrow min\{f\} s.t. slack_f(v) \geq 0$
11: **end for**{ The minimum feasible freq. for each vertex}
12: Set the new frequency, $f_{new} = max\{f_{choose}(v)\}$
13: $\forall v : v'' \rightarrow v$, and v'' is not in this node, send $slack_{f_{new}}(v)$ to v''

Within each node, Algorithm 5 is used to determine its best CPU frequency. It computes the minimum feasible frequencies for each vertex (lines 1-11) in a node, then chooses the maximum from among them as the frequency for that node (line 12), finally propagating this value to all vertices upstream (line 13). Figure 13 illustrates this procedure, with the slacks along the edges represented by S_x's and the execution time of the computation at

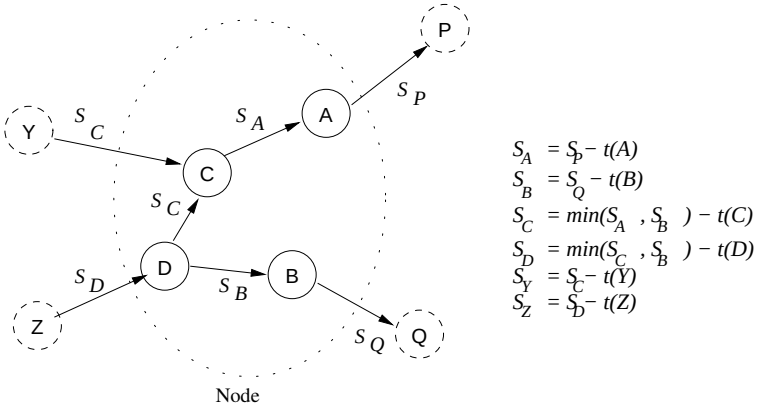


Figure 13: Slack Propagation Within a Node

each vertex represented by t_x's, and shows how the slack is "consumed" by each node and the remainder passed on to vertices upstream.

Being a greedy algorithm, its focus is to quickly reclaim any slack made available, hence it seeks only local optima. As a result, the closer a node is to the sink, the greater will be its energy savings. Fairer methods for slack reclamation would require additional coordination among MSO nodes, thereby introducing additional protocol complexity. The main assumption in this algorithm is that the expected execution time at each frequency for the vertices can be estimated. Research efforts in workload characterization can be used to perform such estimates [40]. Even with accurate application characterization, factors like network jitter, mobility related effects, or even changing application needs can cause changes in slack availability. The response time to such events primarily depends on the frequency at which slack reclamation is initiated. Additionally, since a side effect of the greedy algorithm results in concentrating the bulk of reclaimed slack closer to the sink nodes, this also enables quick reversion of the reclaimed slack, during such conditions.

3.2 Power Tradeoffs and System Implications

In order to effectively manipulate energy usage amongst distributed nodes, it is essential for MSO to understand the power and performance tradeoffs of the underlying hardware. In this section, we present results from detailed power measurements of our evaluation platform which provide the intuition and tradeoffs that drive the system's power management policies.

The hardware environment used in our experiments is the Intel Sitsang platform, with a

PXA255 processor, and 64MB RAM. It runs the Linux-2.4.19 kernel, modified with Xscale and platform specific patches. Each node also has an 802.11b wireless interface, in ad-hoc mode, in addition to a 10Mbps base-T ethernet interface. All power measurements are performed using a Tektronix TDS5104B oscilloscope, Tektronix TCP202 current probes, and Tektronix P6139A voltage probes.

3.2.1 Platform Power Trends and DVFS

The Sitsang platform is designed around a PXA255 XScale processor. The processor supports frequency and voltage scaling via multiple operating points that vary CPU frequency as well as the frequency to the internal PXA bus, thereby affecting latency to memory and I/O devices. The core frequencies available are 400MHz, 300MHz, 200MHz, 150MHz, and 100MHz. Though multiple bus frequencies are plausible for certain core frequencies, for the purposes of analysis and experiments in this work, we always utilize the maximum bus speed possible for a given core speed. This results in five operating points with core/bus frequencies of 400/200, 300/100, 200/100, 150/50, and 100/50. The voltages used are 1.3V for 400MHz, 1.1V for 300MHz, and 1.0V for all other frequencies (the PXA255 electrical specifications prescribe the operating points used, along with the 1V minimum requirement).

For power analysis, we utilize a tunable synthetic workload that has characteristics similar to the robotics applications used in our MSO research [132]. Specifically, for these applications, we find memory access behavior to be an attribute that can vary significantly. Software components like Bayesian classifiers are CPU bound, where performance scales with frequency, whereas image analysis like blob finding displays increased memory activity due to footprints larger than the 32K cache size on the PXA255 processor. In order to provide a fair comparison across this attribute, we have developed a synthetic workload that can be tuned to vary memory boundedness while maintaining the amount of work (i.e. instruction count) performed. This benchmark is used in subsequent evaluations.

When completely idle, the Sitsang consumes 2.5W-2.64W depending upon the operating point to which it is set. A more significant variation can be observed between the different frequencies when active, as illustrated in Figure 14. The figure provides power data when

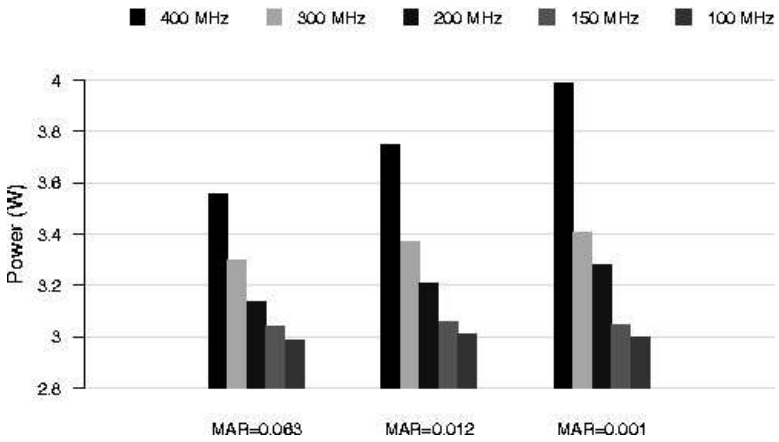


Figure 14: Sitsang Active Platform Power Consumption

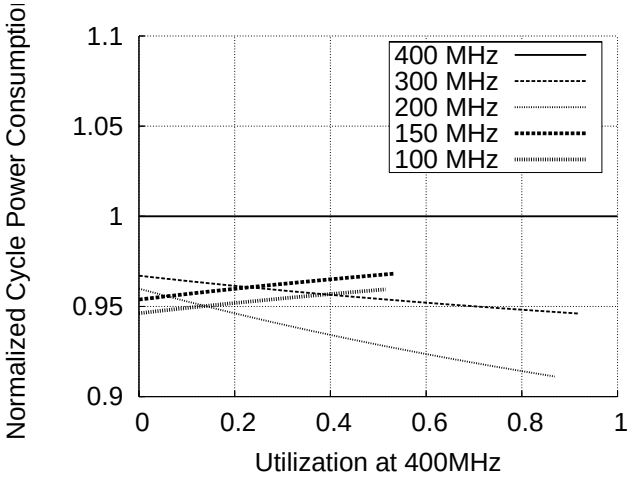


Figure 15: Power Dependence on CPU Utilization, MAR = 0.063

the platform is one hundred percent utilized and executing workloads with varying memory accesses per instruction (MAR), a parameter that in turn affects the cycles per instruction (CPI) required to execute each application. As expected, we see decreasing power consumption as frequency is reduced, with the difference between 400MHz and 100MHz being as high as 25%. These system-level trends underscore the possibilities of energy savings available via DVFS. We also observe from the figure that the system power is not only a function of frequency, but can also vary significantly based upon workload characteristics. Indeed, at 400MHz, the power varies by as much as 11% between the different applications. This highlights the potential benefits and necessity of online monitoring in MSO to dynamically tune energy management for application specific behavior.

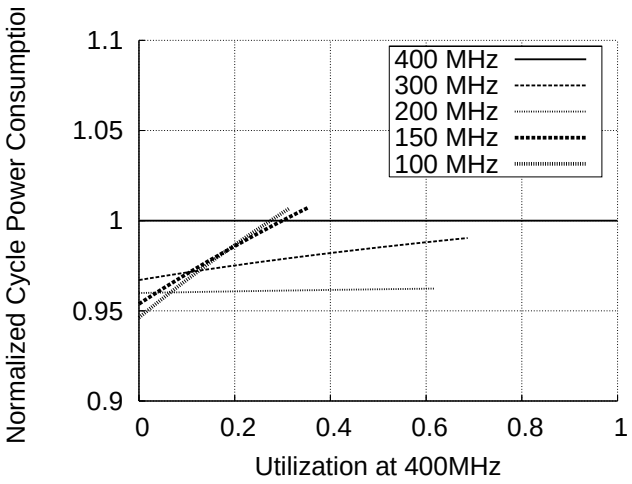


Figure 16: Power Dependence on CPU Utilization, MAR = 0.012

From the results in Figure 14, it is clear that reducing the operating point of our platform, when possible from a performance standpoint, can reduce power consumption. The resulting energy savings, however, are not quite as apparent. For periodic applications, frequency can be reduced when there is slack without a performance penalty. This reduction increases the active portion of a period while reducing the idle period. As recent work has shown, reducing processor frequency may result in reduced power consumption during active portions of the period, but it can also increase energy consumption after some point of slack reclaiming [102, 75]. The existence of these counterintuitive trends can be affected further by the presence of other power management schemes with which DVFS must coexist [52, 103]. We consider these trends by obtaining the *cycle energy*, the combination of the active and idle energy signatures in a period, of the three MAR varying applications across different CPU utilizations in Figures 15, 16 and 17.

Figure 15 illustrates the tradeoffs of the different operating points across utilization behavior for a memory-bound workload with resulting high CPI. As utilization increases, it becomes infeasible to execute at certain frequencies until eventually, only the highest operating point can maintain the performance of the application. Since the application is memory-bound, the performance of reduced frequencies can closely match those of higher frequencies, especially when the bus frequency can be maintained. This is exhibited between the 300MHz and 200MHz operating points as well as the 150MHz and 100MHz frequencies

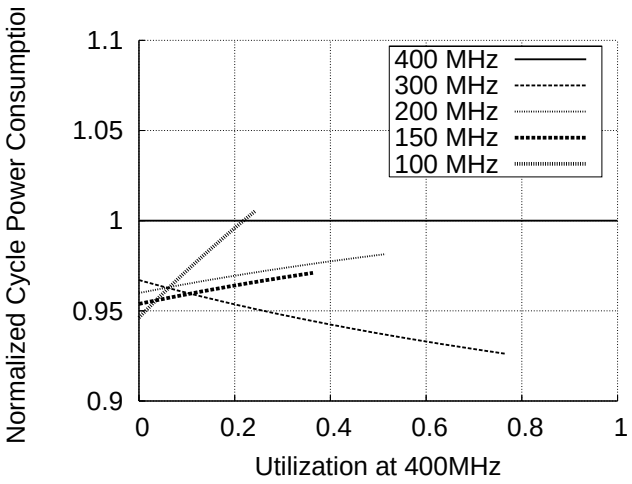


Figure 17: Power Dependence on CPU Utilization, MAR = 0.001

by the fact that the respective energy curves end at about the same utilization (i.e., the modes become unusable at similar load). The reason for this is that the performance of the application is driven by bus frequency instead of core frequency due to the high memory access rate. We can see from the figure that the optimal operating point is only the smallest one possible for very low utilizations, after which 200MHz is optimal even though 150MHz and 100MHz would be options as well. Similar inflection points can also be observed for lower MARs in Figures 16 and 17, though in the latter the optimal operating point gets pushed even further to 300MHz at high utilizations. *These trends show that the energy optimal operating point can vary based upon workload characteristics as well as the utilization required to execute the workload.* It should be noted that even simply monitoring MAR is not adequate (our own results show data stalls per cycle should also be monitored), a full list of required attributes can be obtained via existing workload characterization research [40].

The platform power trends discovered in our experiments directly affect DVFS-based energy management in MSOs. First, they show that when utilizing DVFS to dynamically tune energy behavior via slack reclaiming of periodic applications, MSO middleware must monitor resource utilization information for software components so that it can be aware of 'where' the system is located along the utilization curve, thereby determining the minimum operating point to utilize at a particular node. Second, with online monitoring, MSO can

also determine the performance scalability of an application by determining metrics such as MAR and the resulting data stalls per cycle. By coupling this information with platform power characteristics, MSO can more readily determine application specific inflection points at runtime than can be done by static policies.

3.2.2 Wireless Communication Overheads

In addition to platform energy savings with respect to utilizing frequency scaling, our approach also exploits cooperative systems by offloading computations to take advantage of remote resources and energy reserves. This type of offloading has been shown to provide significant system lifetime benefits in previous work [107]. Here, we continue to leverage this type of energy management by considering the possibility of migrating software components in MSOs during reallocations. A question that arises, however, is how the associated communication energy overheads compare to the benefits of offloading. To obtain insights into this tradeoff, we stream data between two Sitsang platforms over a wireless link. We then monitor the system, CPU, and radio power of one of the systems at different data rates of UDP/IP. These experiments result in the following findings. First, we find that the link becomes saturated at 4Mbps. At this extreme, system power consumption increases by 300mW, while the radio power signature is only elevated by 90mW. The CPU power signature explains this discrepancy, as we observe that the processor is consuming active power during 25% of the time due to packet processing overheads. Therefore, the majority of the power increase can be captured by simply monitoring system utilization. The reason for the minimal increase in radio power consumption, even at high link utilization, is that in ad-hoc mode, the radio cannot be placed into a sleep state—it is always in a promiscuous read mode, the power signature of which varies little from sending. Since the radio power consumption changes negligibly with use, the overheads of utilizing it, for the sake of our flowgraphs with little communication utilization, can be effectively ignored. Therefore, in MSO energy management, we only consider the computational overheads of software components when performing energy load balancing.

3.3 Evaluation

Power consumption at each node is estimated through a daemon that monitors the CPU usage periodically at $0.1s$ intervals, and computes energy consumed based on the CPU frequency and with an application dependent power model (which in turn is obtained by power measurements performed using the oscilloscopes on an instrumented node). Two sets of experiments are performed: to study the effects of energy-aware reallocation, and dynamic resource reclaiming.

3.3.1 Reallocation

The first set of experiments uses two Sitsang nodes running a single application (CPU intensive, with 30% utilization), such that only one node runs at any time, with the other node is idle. By monitoring each others' battery levels, the nodes can cooperatively run the application so that they maximize their battery lifetimes. Overall lifetime of the system is defined as the lifetime of the first node that exhausts its battery completely. Starting from 2kJ for each node, the polling frequency for monitoring is increased, and the corresponding increases in node lifetimes are observed (Figure 18). The results are compared to a static assignment, where one node is always idle and the other is always busy. As we increase the frequency of reallocation, the system lifetime increases, but more time was also devoted in performing the reallocations than performing the actual work. The net yield thus shows an increase, followed by a decrease, with the highest increase found to be roughly 11.5%

Next, we study an example on a five node testbed, with the topology in Figure 19. The application flow graph is chosen to accommodate all three different kinds of event flow combinations possible, i.e., linear, split, and join flows. Each node runs an instance of MSO, and begins with a fixed energy level. The threshold for reallocation decisions is set to 50J, and the frequency of polling (for monitoring neighbors' energy levels) is chosen to be 25 sec. All application components are CPU bound, and while two of them run at 80% CPU utilization, the other three run at 15%. These values are deliberately chosen, to allow an imbalance in processing, among the nodes. Such a workload is representative of applications that perform different pipelined processing on data. For instance, in an image processing

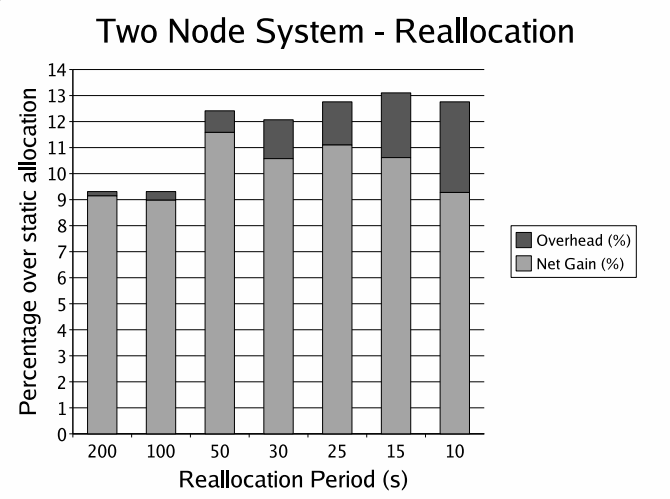


Figure 18: Reallocation in Two Nodes

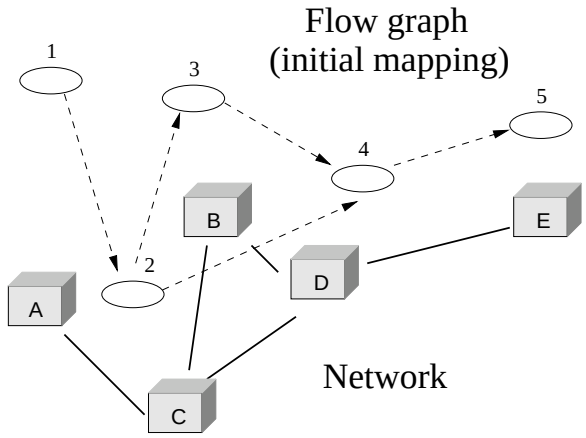


Figure 19: Experimental Setup

pipeline in the robotics domain, the pre-processing stages like edge detection, scaling, etc. are data intensive, whereas applying learning algorithms on the images are typically CPU intensive. Events of size 4KB are sent via a single source at the rate of 1 per 1.5 seconds. For this experiment, we ignore the timeliness of event delivery (causing the check in line 6, in algorithm 3 to always return true). This enables aggressive relocation of vertices to the best available neighbors. We discuss the benefits of using MSO along two parameters: (i) the system lifetime of a cooperating group and (ii) the parity in the lifetimes of the nodes forming the group. An example of the first type of requirement is in collaborative tasks that require the participation of all mobile nodes. The second rule can be useful in enforcing a uniform policy for all participants in the task. For our purposes, we term the lifetime of the first dying entity in the group as the system lifetime, and we quantify lifetime parity by measuring the standard deviation of individual node lifetimes. We observe the trend of these metrics, as the initial energy available with the nodes are varied. We compare the results of our reallocation with a static assignment (Figure 20). The difference in the lifetimes afforded by these strategies increases, as the initial energy increases, with the differences being close to 10% at 5kJ. This is a consequence of the fact that the lifetime of the node(s) executing the computationally intensive components of the application flow graph exhibits a linear relationship with the initial energy. However, as reallocation shifts the heavy computation among all the nodes, this effect is mitigated. Similarly, the lifetime parity with reallocation shows no particular trend with increasing initial energy (Figure 21), as against the widening gap in node lifetimes observed with a static allocation.

3.3.2 Effects on Applications

Next, we study the effects of MSO-based energy-aware reallocation on an example application. We consider a network of five Sitsang nodes in a linear configuration, and map an image processing pipeline consisting of (i) blob detector, followed by (ii) grayscaling, and finally, (iii) edge detection. Images from a source (such as a camera) are sent to the blob detector as events roughly every 1.2s (in order to obtain 70% utilization when all the components are co-located), which then get processed in the pipeline. Simultaneously, a robot

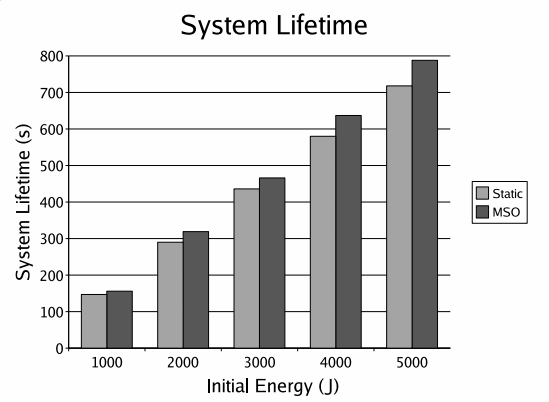


Figure 20: Application Lifetime

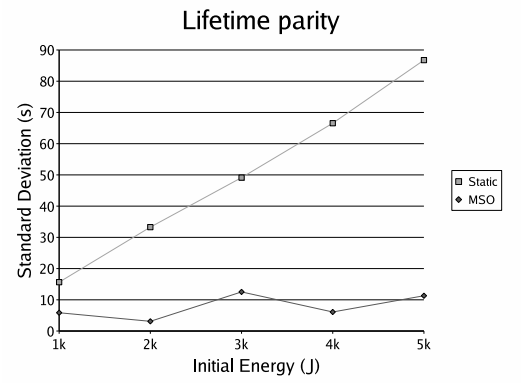


Figure 21: Node Lifetime Parity

Table 4: Application Lifetime vs. Performance

	Non-Cooperative	Cooperative	Gain (%)
Lifetime	294s	336s	14
Image events σ	20.7ms	225.3ms	988
Navigation events σ	19.3ms	90.1ms	366

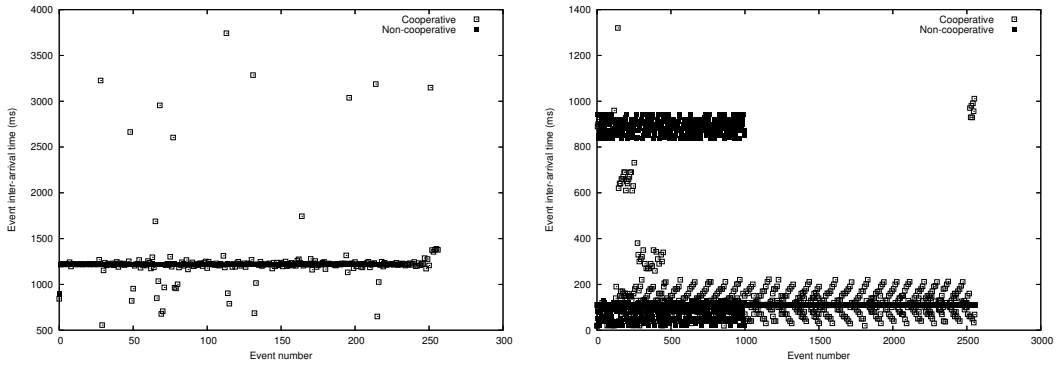
navigation pipeline consisting of a localization module, which computes the location of the robot from its odometry data, and a navigation module, which uses the location to send drive commands in order to reach a predetermined destination, is also run in the first node, with the rationale that while the first robot navigates to its destination, the other robots will follow this leader in a linear fashion. Starting from 2kJ, the application lifetimes, as well as the event inter-arrival times with energy-aware reallocation is compared against a static allocation (cooperative vs. non-cooperative), and the results are shown in Table 4.

A gain of about 14% in application lifetime is achieved with our scheme, however, the event inter-arrival times suffer high variations, as can be seen by the standard deviations shown in the table. This results from differing hop counts as events get processed through the chains, arising from relocation of the application components.

A second experiment performed under an identical scenario, replacing the linear network with a network forming a complete graph, shows a similar behavior (Figure 22). In this case, however, variations in the cooperative case are actually lower, with the navigation events. This is due to the fact that, as offloading other components decreases the utilization of the node running the navigation pipeline, events get processed at more regular intervals.

3.3.3 Dynamic Resource Reclaiming

The next set of experiments evaluate the dynamic resource reclaiming algorithm, over a synthetic workload. As discussed previously, we apply this technique to source-defined event slack available at the application sink node. The setup for the experiment consists of five Sitsang nodes in the same configuration as in the previous study. Each vertex executes a memory-intensive synthetic application. The application is run at three different scenarios, corresponding to having a CPU utilization of 40%, 60% and 80%, by increasing



(a) Image pipeline

(b) Navigation pipeline

Figure 22: Offloading - Application Results

the duration of execution of each event in the application. Events of size 8 KB are sent with a periodicity of 2.5 seconds. The execution time of each event, when run at 80% utilization and the highest frequency (400 MHz) is found to be roughly 2s at each node.

The resulting energy consumption of the application is measured in the presence of the resource reclaiming algorithm. This figure is compared against a choice of frequencies for each possible slack value that is optimal in that it minimizes energy consumption. The energy values are normalized against the default case where all nodes run at the highest frequency. As shown in Figure 23, the algorithm is found to closely track the most optimal settings. In some cases, especially at the top frequencies, our algorithm appears to outperform even the optimal solution, but this is only because of missed deadlines, i.e., the algorithm reclaims more slack than available, thus saving more energy but hurting performance. This is due to errors in predicting the execution time at various frequencies, and other sources of error in event delivery. For the higher utilization workload, as both the slack available, as well as the execution time are high, any errors in estimation can cause large deviations from the optimum frequency settings.

Finally, we evaluate the overheads of various strategies that can be employed for resource reclaiming. We consider three strategies in this study: (i) Aggressive, where slack is polled frequently (every 30 sec), and positive as well as negative slacks are immediately propagated throughout the network, (ii) Conservative, which polls for slack less frequently (60 sec), and

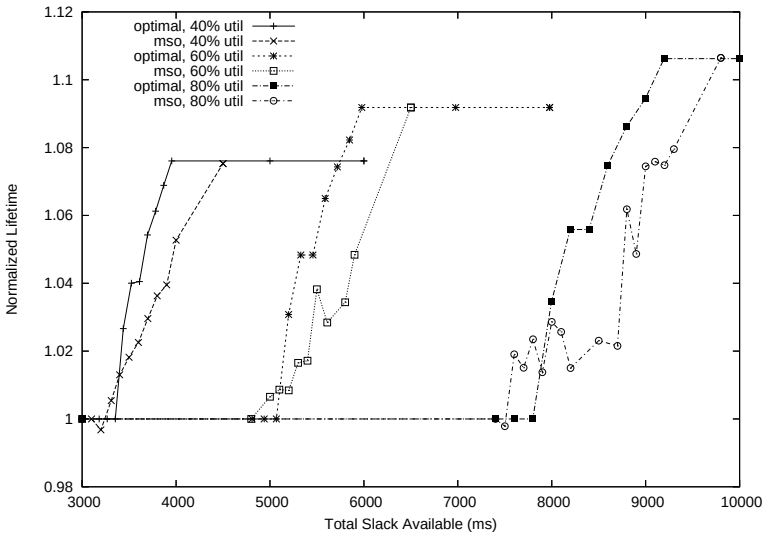


Figure 23: System Energy Savings With Slack Reclaiming

Table 5: Slack Reclaiming – Strategies

Strategy	Mean slack (ms)	Mean |slack| (ms)	Reclaims
None	1188	1204	0
Aggressive	-23	305	1012
Conservative	59	245	528
Opportunistic	-45	215	822

(iii) Opportunistic, where positive slacks are propagated at a low rate (60s), and negative slacks at a high rate (30s), so as to conserve energy without a high overhead, but react relatively quickly when performance is hurt. The event traffic was such that all the ranges of slack values were used in the test, over a period of about 15 minutes. The results of these strategies are shown in Table 5.

The aggressive strategy is more prone to mispredictions and overcorrections, than the others, but it is able to utilize all of the slack. The conservative approach is relatively more stable, with almost half the number of reclaims, but it can let the slack available for shorter periods of time go unrecovered. The opportunistic strategy strikes a balance between these extremes, to react quickly during performance critical phases alone. Correspondingly, the number of slack reclaims also lies between these strategies.

3.4 Discussion

As mentioned previously in Table 1, modern wireless technologies allow much higher data rates, and this trend is poised to cross even 4Gbps soon, with WirelessHD. This increases the opportunities to perform offloading, since communication costs are expected to go down (for instance, wirelessHD targets mobile devices). A heterogeneous computing environment in the next generation of pervasive systems also provides us with choices in mapping portions of the application to specific platforms, resulting in more flexibility and choice. Previous work has shown that for mobile applications, use of memory in a remote machine can be more power-efficient than local memory, under certain scenarios [55]. This has the potential to increase the level of cooperation among nodes, by decomposing the distributed application at a much finer granularity. Currently, the division is fairly coarse in order to minimize the volume of data flow between components. The rationale for this is further supported by trends in server systems indicating that over 40% of system power consumption is due to the DRAM memory, and nodes with more power-efficient memory can address the memory needs of other nodes in the network.

Slack reclaiming is used for energy management purposes in this work, but can be used in other tradeoffs that balance application performance with resource saving. This can apply to co-processors as well as devices, in increasing their idle periods. The energy aware reallocation method described in this chapter can be applied to multi-core systems as well, as it provides a low overhead mechanism for completely decentralized energy management.

CHAPTER IV

ENHANCED DEVICE SERVICES IN VIRTUALIZED SYSTEMS

Prior chapters describe the role of middleware in presenting services to distributed applications, that allow them to be run on MANETs. By sharing the underlying resources, the middleware allows cooperative execution. With the increased interest in virtualization and the generic resource sharing capabilities it enables, we identify the opportunity to use similar middleware-based techniques to share device functionalities in such environments. This is especially useful in mobile environments, as platforms in this domain are typically required to support a diversity of devices.

In this chapter, we explore sharing opportunities and methods for multimedia devices, the goal being to make it easy to dynamically compose, share, and use these devices to provide efficient multimedia services. Termed VMedia, our method enables media-rich applications by better supporting flexible access to and use of the many media devices present in today's systems. Specifically, VMedia offers new hypervisor-level support both (1) for efficient and flexible device sharing and (2) for dealing with and exploiting device differences and diversity. We then present an extension to VMedia, termed CustomCam, that allows customization of a multimedia device to particular uses.

In virtualized systems, a workload can be run in its own isolated container, called a virtual machine (VM), each with its own runtime components, including operating and file systems. A Hypervisor or Virtual Machine Monitor (VMM), in conjunction with one or more privileged VMs, termed 'Service VM' or 'Domain 0' (Dom0), implement virtual instances of physical resources, such as CPU, memory, and I/O devices. Constituting a virtual platform for the VM, these virtual resources are multiplexed over the physical resources existing on the machine. Virtualization of basic resources like CPU and memory is implemented by the low-level hypervisor or VMM resident on the machine, whereas the Service VM is responsible for virtualizing devices. Using a Service VM allows ready utilization of device

drivers for the multitude of I/O devices employed by VMs. For high end I/O devices, the functionality of the Service VM may also be provided directly by the device itself, in the form of self-virtualized devices [123, 91]. VMedia is such a service VM based approach to virtualize multimedia devices.

Device virtualization is a key element of virtualized systems. A simple, non-intrusive method is to create a virtual device that emulates a physical one. In this case, the virtual platform provides I/O resources (configuration registers/memory) just like the physical platform, and the guest OS interacts with the virtual device in the same fashion as it did with the physical device, using its own device driver. However, this approach has inherent performance limitations, because device emulation requires fine-grain involvement from the VMM and/or Service VM (i.e., at the level of memory/register access). As an alternative, all current system virtualization solutions provide simpler virtual I/O devices, which present different access interfaces to guest VMs, such as shared memory circular buffer rings, rather than I/O memory and registers. Device drivers hide these interfaces from the guest OS kernel, providing it with standard device interfaces. For example, a virtual NIC device driver provides an ethernet interface that is identical to the interface provided by the physical NIC's device driver. Using these simpler virtual I/O devices and the corresponding device drivers provides substantial performance benefits compared to the emulation approach. Above this layer, guest operating systems, then, operate just like in non-virtualized environments, using their device drivers and other internal functionality to present applications with efficient higher level system abstractions like sockets, files, etc.

4.1 Multimedia Device Virtualization

Researchers and developers have already recognized that standard virtualization methods like those described earlier have limitations. Consequently, they have introduced the notions of para-virtualization [23] vs. full virtualization (e.g., through device emulation), where device drivers still export standard APIs, but are handled by the VMM to implement the appropriate functionality [139]. Further, more recently, researchers have introduced the notion of application virtualization [17] where a library operating system extends the

Exokernel idea [50] of providing customized operating system to applications, thus delivering only the functionality needed by the applications. Libra provides services required by a Java application running within a JVM, by implementing frequently accessed services locally and relying on the hypervisor for other, less frequently used services. This opens interesting avenues for new investigations and for exploring opportunities for improved performance or even functionality, such as combining multiple devices, as discussed in Section 4.5.

This work addresses these questions for multimedia systems and applications, by developing and experimenting with the *VMedia approach* to I/O virtualization. This approach exports to applications *logical devices* that are semantically enhanced versions of the physical devices present in the underlying platforms. Specifically, a VMedia logical device has attributes and provides access methods that go beyond defining "what the device is", as in current systems, to also define "how it is used". For example, a logical camera device might provide a rich multimedia access interface, like Video4Linux [13] (V4L), instead of the low-level API presented by a USB camera. Another example is an iSCSI-capable NIC [6], which provides both a SCSI access interface for block devices and a normal ethernet access API.

Previous research has already demonstrated the utility of using logical rather than physical device interfaces. In our own work with V4L, for instance, we have shown that this interface can be used for transparent access to both local and remote physical camera devices [82]. The VMedia approach exploits I/O virtualization to go beyond such transparent device remoting. It provides a service-based interface to media devices, in order to (1) allow sharing of these devices, and (2) make it possible to dynamically create from physical devices virtual ones with different properties and capabilities. We note here a similarity in approach between VMedia's service-based logical devices and modern file system services, such as NFS [8] and GPFS [129], provided by today's network storage solutions. Such file services can be seen as a logical device, which are provided in addition to block-based virtual disk devices (e.g. devices supporting SCSI interface). Utilizing filesystem level abstraction, these storage logical devices allow sharing of content (files) in a straightforward manner, while the usual block-based virtual devices do not.

Previous work has also shown the utility of using semantic information to enhance certain physical devices, as with smart disks [137], for instance. However, for cost reasons, these solutions have not been widely popular. VMedia addresses this issue by using software to enhance the virtual platform, rather than requiring new or extended device hardware (e.g., expensive device controllers). Furthermore, the service-based virtualization used by VMedia affords several additional advantages.

First, it can simplify the guest VM's OS kernel without sacrificing any of the functionality presented to applications. Second, by using domain-specific semantic knowledge, I/O virtualization at higher levels like V4L can provide better performance than solutions operating at the device level. Third, the use of logical devices can provide better opportunities for consolidation in the Service VM, based on information from multiple guest VMs. Fourth, a logical device may provide better performance and/or more functionality than that offered by a single physical device, by having the Service VM use an ensemble of physical devices to realize the logical device, for example. Shifting logical functionality to the Service VM also frees up computational resources at the guest VM side. A guest VM can then use these resources to implement other useful functionality. It may also increase the platform's scalability in terms of the number of VMs it can support. Shifting computations related to I/O also allows guests to function better in the presence of resource restrictions, such as limited availability of cores or licensing restrictions imposed by software for certain number of cores.

In summary, this work presents the VMedia framework for logical devices, focused on the multimedia domain:

- VMedia is used to export a 'multimedia' device to guest VMs, using the standard Video4Linux [13] interface.

- The multimedia device is implemented with software running in the Service VM, Dom0. By acting as a 'hub' for such logical virtual devices, the Service VM can provide enhanced multimedia services to guest VMs, with efficient and flexible device sharing, and offering new device capabilities.

4.2 VMedia Design and Architecture

VMedia Design. Unlike network and storage devices, which are virtualized via time- and space-sharing respectively, the rich semantics associated with multimedia devices make sharing at the device level more difficult. Web cameras and microphones, for instance, can be time-multiplexed among multiple VMs, but arbitration of the device will be difficult. For example, different VMs may want to change the attributes of the device in mutually exclusive ways. This means that the virtualization system must maintain a 'context' for the device per VM, and change the device to a particular context whenever the corresponding VM requests access. Additionally, in sensor devices such as cameras, sharing can be achieved more efficiently by replicating sensed data (e.g., frames) to data consumers, rather than using time-sharing techniques. Current virtualization solutions ensure that multimedia devices are used exclusively by one VM. Virtualization of these devices is done at a lower level, such as USB and PCI, and access is provided to a single VM as a passthrough.

The VMedia framework creates enhanced opportunities for sharing, by implementing logical devices that are accessed via a standard multimedia API, which is Video4Linux (V4L). Guest VMs' device drivers interact with the VMedia Service VM using a higher level API, again similar to V4L. VMedia's *virtual multimedia device* thus exported has several interesting properties. First, such a device need not be a simple mapping of the physical device that is being virtualized. In fact, additional interesting properties of a virtual device can be entirely implemented in software, an example being a virtual device that supports multiple palettes and image resolutions, while the physical camera supports only one of these. Second, device implementations can be entirely dynamic, using runtime code generation and extension techniques [56] and placing such extensions into Service VMs for shared use by all/some logical device users. Extensions may implement data transformations, for instance, to guarantee certain privacy constraints on the data captured by the device [82] or to provide data to end user applications in certain forms. Third and as explained next, multiple guests can efficiently share VMedia's multimedia devices, via its *MediaGraph* abstraction, described in detail in Section 4.2.3. The outcome is that a guest VM can be oblivious to how the physical device is being accessed, and that end users need

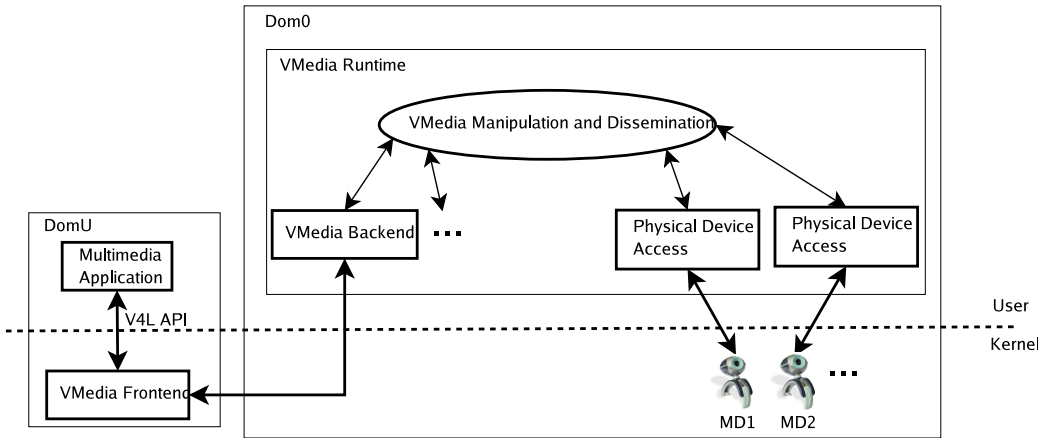


Figure 24: VMedia Architecture

not rely on complex applications hosted by guest VMs for such purposes.

The VMedia design also makes it possible to compose new virtual devices from multiple, possibly heterogeneous physical devices. For example, by using two similar camera devices and with appropriate phase lag, it is possible to support twice the frame rate than what could otherwise be afforded by a single device. As another example, a context sensitive camera device can be created using a camera and a microphone where an image is only captured in the presence of sound, else returning an image from a cache without capturing a new physical image.

VMedia Architecture The VMedia framework consists of two main components: (1) virtual multimedia devices and associated drivers running in guest VMs (client side), and (2) the VMedia runtime that executes in the Service VM, or Dom0 (server side). The VMedia runtime accesses the physical multimedia devices and provides guest VMs with access to the media data via virtual devices. Figure 24 depicts a high-level overview of these components.

4.2.1 Client Side Components

Client (Guest VM) side components include a virtual multimedia device and the corresponding kernel device driver. The virtual multimedia device is an extension of the virtual interface (VIF) abstraction presented previously [123]. The device is assigned a *unique ID*

and consists of two message queues, each of which is a circular ring buffer. One message queue, called the *send queue* is for outgoing messages to the physical device, sent from the guest VM to the VMedia runtime. The other queue, called the *receive queue* is for incoming messages from the device, sent from the VMedia runtime to the guest VM.

A pair of signals is associated with each queue. For the send queue, one signal is intended for use by the guest VM, to notify the VMedia runtime that the guest has enqueued a message in the send queue. The other signal is used by the VMedia runtime to notify the guest domain that it has received the message. The receive queue has signals similar to those of the send queue, except that the roles of guest VM and VMedia runtime are interchanged.

The kernel driver, called the VMedia frontend driver, registers a V4L device with the guest VM kernel. Applications running in the guest VM access the V4L device via V4L specific ioctls or file access system calls (e.g. read). These calls are converted into VMedia messages by the frontend driver, and sent to the other end via the send queue, where the backend component of the VMedia runtime receives them and performs appropriate actions. In response to these messages, the VMedia runtime may generate messages for guest VMs, which are received by the frontend via the receive queue. These messages in turn are mapped to application-specific calls.

These messages do not carry media data themselves. All media I/O takes place via a pool of shared memory buffers shared between the guest VM and the VMedia runtime. These buffers can also be mapped directly in application address space, thereby allowing I/O with minimal copying.

The virtual devices we have implemented to date are those focused on the multimedia domain, supporting properties related to a video capture device, such as image size, image depth and palette, via the V4L interface. Properties for devices other than video, e.g. audio and VBI, can also be provided via this interface, and this is part of our future work. Virtual devices also support some VMedia-specific logical properties, such as orientation and quality, exported to applications via an extension of V4L API. These properties, along with the multimedia properties discussed above, are used by the VMedia framework to

compose efficient and enhanced virtualized I/O solutions. Improved performance coupled with transparency to applications and to the guest VM's operating system are the potential outcomes of this approach, as shown in more detail in Section 4.5 below.

4.2.2 VMedia Runtime

The VMedia runtime realizes the self-virtualized I/O abstraction [123] with software resident in a Service VM. The runtime is responsible for:

- scalable and isolated multiplexing/demultiplexing of a large number of *virtual devices* mapped to one or more physical devices;
- providing a lightweight API to the hypervisor and guest VMs for managing virtual devices;
- efficiently interacting with guest VMs via simple APIs for accessing the virtual devices; and
- implementing multimedia domain-specific extensions that enable semantically enhanced logical virtual devices.

These functionalities can be broadly categorized as 'management' and as 'I/O virtualization'. For a virtual multimedia device, management functionality is provided to the hypervisor and to the guest VM using the device. In addition to obvious management actions like device creation and removal, the VMedia runtime provides additional, domain-specific reconfiguration functionality. For example, a video capture device may allow changes in image properties, such as colormap (color or grayscale), image depth and image size itself. The application running on the client side may request these changes, which in turn are sent to the VMedia runtime as management actions by the client side driver. The runtime makes appropriate changes in the properties associated with the virtual devices, along with any changes that may be necessary related to the I/O processing in order to satisfy these. For example, if the image size requested of a virtual multimedia device is different than that of physical device, an appropriate scaling filter may be installed in order to meet this mismatch. These reconfiguration actions are discussed in detail in Section 4.2.3.

The key functionality of the VMedia runtime is to implement I/O virtualization via sharing of physical multimedia devices among multiple virtual devices. The runtime utilizes semantic knowledge of virtual devices in order to perform this sharing. Since the runtime knows about the multimedia properties of the virtual device, e.g., the direction of I/O (input vs. output), type of content (such as image and audio), information about content (such as image size and colormap), it can use these properties in order to build an *information* flow from physical devices to virtual devices. For example, for input multimedia device, such as cameras, *images*, rather than *bytes*, are sent to virtual devices.

The VMedia runtime is composed of multiple entities that jointly realize the functionalities described above. These entities can be categorized broadly as (i) Physical Device Access, (ii) Virtual Device Backend, and (iii) Media Manipulation and Dissemination.

Physical device access entities implement the media device-specific methods for obtaining media data from or sending media data to the physical device, one entity per device. For example, the data could be obtained from the USB based camera via a V4L-based device driver, or it could be obtained over the network if the camera is attached to a remote device, such as a cellphone connected to the host system via USB, bluetooth, or wireless. Depending on the type of device and how it is connected to the host system, the latency and throughput of media data will vary.

Corresponding to every guest VM frontend, the VMedia runtime contains a *backend* entity. These form a point-to-point connection with the frontend, and merely work as a gateway of information from (to) guest VMs to (from) VMedia runtime.

For input devices, such as cameras, captured media data from device access entities is provided to the VMedia manipulation and dissemination component (VMediaMD), where this data is transformed if required and is disseminated to the virtual device backend(s), which then flows to the guest VM frontend. For output devices, such as speakers, media data received from the guest VM is provided to the VMediaMD, where it is transformed if required and is provided to the appropriate physical device access entity for output. Currently, the VMedia framework only supports input devices, and hence, the remainder of this discussion is limited to these devices only.

The control flow for input multimedia devices (e.g., image capture requests and property changes) is similar to that of media data flow for output devices, with some exceptions. Management control requests may change the VMediaMD component itself. For example, if a virtual camera device requests a grayscale palette, the VMediaMD component may need to add another component to provide this functionality. Further, depending on the sharing of physical devices, if there is a common property/functionality required by all virtual devices and if it can be directly provided by the hardware, this control flow may reach the physical device access components themselves. Some of the management control decisions may only be taken at the service VM level itself, such as the orientation of the physical device.

I/O control requests go through a minimal path of the VMediaMD component, mostly providing arbitration. Arbitration decides which of the physical device access entities should receive this request (there may be more than one). VMedia-specific logical properties can also be used for arbitration. For example, a guest VM may indicate its preference for a certain viewing area via *orientation*. The arbitration logic matches this preference to one or more physical devices. Arbitration also decides whether it is necessary to forward a request to a physical device, since it may already be involved in the I/O. The request is only forwarded if it is not.

Media sharing in VMedia is governed by a simple arbitration principle – any request received from the guest VM during the time when a media capture I/O is pending can be satisfied from the result of this capture. Hence, if multiple VMs issue capture requests simultaneously, the capture is performed only once and the result is distributed to all VMs. This type of device sharing is a special case of space sharing, where a device can be shared by all virtual devices at all times due to the *semantic* properties of the device.

4.2.3 The MediaGraph Abstraction

MediaGraph forms the core sharing infrastructure of VMedia. Implemented over an event-based overlay network, it handles each multimedia frame as an event, propagating it through the network. Appropriate transformations are performed on the events as they flow through

the network. Finally, events exiting the network are fed to the appropriate VMs.

Abstractly, the VMedia runtime entities described above and the control and data flows implemented by them form a di-graph structure, termed MediaGraph. This graph is built to meet the properties specified by end user applications for the virtual multimedia devices they are using. Specifically, the MediaGraph implements efficient media dissemination by consolidating common computations and by reducing communication costs via data filtering. Moreover, the MediaGraph abstraction is dynamic – it can be modified when new virtual devices are added and/or when the properties of existing multimedia devices are modified. Such modifications are triggered by configuration events generated by guest VMs and/or by monitored changes to devices.

Physical device access entities and virtual device backends form the edge vertices of the MediaGraph (sources and sinks, respectively), whereas VMediaMD entities form the internal vertices. These internal vertices correspond to various arbitration and transformation functions. Transformation functions perform the necessary conversions from the media format provided by the physical sources to formats desired by the backend at the guest VMs, and directed edges in the MediaGraph represent the control and data flows.

A sample MediaGraph is shown in Figure 25. As seen in the figure, the cameras, represented by the source nodes S_1 and S_2, generate image frames, that are then sent to the transformation nodes T_1 through T_4, that perform transform operations on the images and send the final outputs to the backend nodes K_1 through K_4, which provide the processed images to the multimedia guest VMs.

The MediaGraph abstraction enables efficient sharing of the multimedia content by avoiding redundant transformations that may be required by multiple sink nodes (backends) in order to support the properties. For instance, the graph shown in Figure 25 combines the common transformation T_1 for backend K_2, K_3 and K_4, thereby reducing the overall cost paid by the VMedia runtime. Such transformation sharing by structuring components in a di-graph is not unique to MediaGraph, similar approaches have been considered elsewhere [109, 148]. Next we describe an algorithm to maintain the efficient sharing when a sink node is added or deleted for a MediaGraph containing single source node.

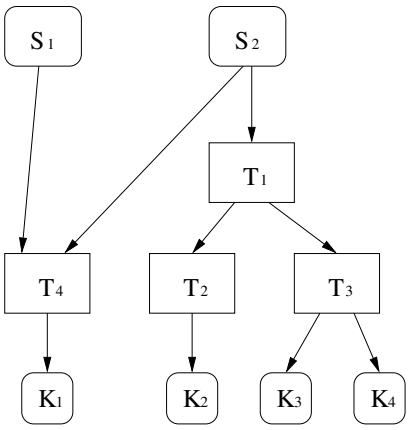


Figure 25: An Example MediaGraph

Maintaining the MediaGraph for Efficient Sharing of a Single Source The amount of computation required to carry out the transformations in the MediaGraph is dependent on the topology of the graph. For instance, if the source image needs to be transformed into a grayscaled and scaled down image in order to serve a VM, performing the grayscaling prior to the scaling down operation involves extra computation than performing the other way round (since the grayscaling is applied on a larger image). Further, the structure of the MediaGraph is dynamic, due to addition/deletion of sinks, and also due to changing content parameters (e.g., resolution, color depth, etc.). In this work, we use a greedy algorithm to build and maintain the MediaGraph, in response to changes in the sinks.

To select the sequence of transformations to be performed in the graph, we group its transformation actions along the various parameters that define the content, and further define an ordering relation among the parameters within each group. For any two parameters a and b within a group, we define the ordering relation $a < b$ if the information content in b is more than that in a. Using this relation, for instance, we can define the depth of the image with the ordering 8bpp $<$ 16bpp $<$ 32bpp. The groups themselves are ordered in the graph in such a manner that data reducing operations (like scaling down, for instance) are closer to the sources than data increasing operations, in order to minimize the computation costs. The multimedia content at any node can thus be represented using the n-tuple $< a_1, a_2, ..., a_n >$, with each tuple corresponding to a group, and $a_1, a_2, etc.$ each

represents the parameter within its group. As discussed previously, the groups themselves are ordered in non-decreasing order of computational complexity, to minimize the amount of processing carried out in the transformations. An example 3-tuple is <resolution, depth, grayscale/color>.

Algorithm 6 Addition of a sink

1: Let the sink's required parameters be $< r_1, r_2, ..., r_n >$, and the source parameters be $< s_1, s_2, ..., s_n >$.
2: **if** $\exists k, 1 \leq k \leq n$, such that $r_k > s_k$ **then**
3: Change the source's content to match $< max(r_1, s_1), max(r_2, s_2), ..., max(r_n, s_n) >$, where $max(a, b) = a$, if $a > b$ and b otherwise
4: **for all** $j, 1 \leq j \leq n$ **do**
5: $s_j \leftarrow max(r_j, s_j)$
6: $\forall$ transformation node N if $input(N) \neq s_j$, change the transformation so that $input(N) = s_j$
7: **end for**
8: **end if**
9: $N \leftarrow sourcenode$, $j \leftarrow 0$ {Now, $\forall k, 1 \leq k \leq n, r_k \leq s_k$}
10: **repeat**
11: $parentnode \leftarrow N$, $j \leftarrow j + 1$
12: **until** $\forall N$, such that $parent(N) = parentnode$, $output(N) \neq r_j, \forall j, 1 \leq j \leq n$
13: $currentnode \leftarrow parentnode$
14: **for all** $j, 1 \leq j \leq n$ **do**
15: **if** $r_j < s_j$ **then**
16: Create a transformation node T, such that $input(T) = s_j$, $output(T) = r_j$
17: Connect $currentnode$ to T, such that $parent(T) = currentnode$
18: $currentnode \leftarrow T$
19: **end if**
20: **end for**
21: Connect sink to $currentnode$

When a virtual camera is opened by a process in a guest VM, this action is translated to an addition of a sink corresponding to the VM, to the MediaGraph. Conversely, when the virtual camera is closed by the process, or if the guest VM is destroyed, the corresponding sinks are deleted from the graph. On the addition (Algorithm 6) of a sink (with desired content parameters $< r_1, r_2, ..., r_n >$) to a source node with parameters $< s_1, s_2, ..., s_n >$, a check is performed to see if any of the desired parameters exceed the source parameters (line 2), and if so, the source parameters are updated to the maximum of the existing and the desired parameters (line 3), with all other transformations updated accordingly (lines 4-7). Next, the graph is traversed starting from the source node to find a maximal match of

the desired parameters among those of the existing transformations (lines 9-12), and finally the sink is connected to this node via necessary transformations (lines 13-20).

Deletion of a sink (Algorithm 7) begins with removing the sink node from its parent (line 1), and then traversing towards the source node until all unnecessary transformation nodes (those that serve no other nodes) are removed (lines 2-3). Finally, it is determined if the source's parameters can be lowered due to the removal of the sink, (lines 5), and if so, carried out (lines 6-11).

Changing a sink's parameters results in the actions of the deletion of the sink node from the MediaGraph, followed by an addition of a sink node with the new parameters.

Algorithm 7 Deletion of a sink

1: Disconnect sink from $parent(sink)$, set $currentnode \leftarrow parent(sink)$
2: **while** $|children(currentnode)| = 0$ **do**
3: Delete currentnode, set $currentnode \leftarrow parent(currentnode)$
4: **end while**
5: **if** $currentnode = sourcenode$ **then**
6: **while** $|children(sourcenode)| = 1$ and the child is not a sink node **do**
7: Set $N \leftarrow child(sourcenode)$
8: Change source's parameter s_j to $output(N)$ for appropriate j
9: Delete node N
10: $\forall T$, such that $parent(T) = N$, connect sourcenode to T
11: **end while**
12: **end if**

The middleware-based approach for implementing sharing in VMedia also makes it easy to extend the implementation to sharing of devices across distinct physical machines. As a result, remote device sharing and local device sharing are treated by higher levels in an identical manner. Section 4.5.5 presents an example of using a remote device via VMedia and CustomCam (described in the following pages).

4.3 CustomCam

CustomCam [111] addresses device sharing and interoperation in the mobile domain, where challenges for device emulation and extension are due to the diversity of actual devices, of device capabilities, and of the contexts in which they operate. Additional challenges in the mobile domain are caused by the fact that virtual machine migration, an important

function in virtualized systems, will be difficult [121] without functionality in addition to that provided by commercial products for data centers [150]. For instance, an approach that allows VMs to maintain a consistent view of physical devices by presenting only a common minimal set of device features is too limiting and worse, it cannot account for changes in the environmental conditions or contexts in which devices are used.

Device sharing and interoperation in mobile systems requires models and interfaces with which virtualization infrastructures can dynamically change device functionality to account for runtime changes in device use, conditions, and contexts. Hence, CustomCam implements the safe, runtime extension of device functionality, by enabling the virtualization layer, guest operating systems, or applications to safely specify and deploy custom code that runs in conjunction with device accesses. This approach entails a number of benefits, as detailed below:

- It provides a uniform view of devices to VMs, independent of the physical host. For instance, a TCP network device may be implemented in software, by providing the TCP stack to the virtualization layer, or in hardware, by exploiting the TCP offloading functions of an enhanced network interface [161]. The VMs using such a device need not be cognizant of how the TCP device is realized.

- It can be tuned to available hardware resources. For instance, in the previous example, a node lacking a network interface supporting the offloading feature may instead have a coprocessor that can be used to implement the feature [123], without the VM being aware of this fact.

- It enables efficient use of remote devices. For instance, an imaging application that requires only those frames captured by a remote camera that contain specific features can provide feature selection code that determines said features. Such an extension can use this code to filter traffic, providing only the frames of interest to the application, thereby reducing communication and computational overheads experienced by the application and the platform on which it runs [34].

- It opens up new possibilities for sharing device functionality. Extending the previous

example, when multiple VMs are interested in the same image features, common data can be shared at the device driver layer itself.

A known problem associated with the runtime extension of system-level software by applications is safety [27], where faulty or even malicious code injected by applications into the system layer can affect the behavior of other VMs that rely on the system. In our implementation, we use the split device driver model afforded by the Xen hypervisor, on which we implement an overlay-based middleware framework in the device backend (i.e., in the device driver layer). This middleware permits the runtime extension of overlay functions by allowing arbitrary code fragments to be included into the overlay, i.e., into the device feature set, at any time deemed appropriate by the application. Safety is attained by use of dynamic code generation, where code fragments specified in a safe subset of the C language [47], and runtime binary code generation guarantees that such code cannot damage other elements of the device driver layer.

The EVPath middleware used by VMedia backend makes it possible to compose higher level services or applications as sets of computations that can filter events, forward them, or transform them. These computations are specified in ECL (a subset of the C language supporting loop, conditional, and return statements), and hence they can serve the needs of individual applications and diverse devices. Further, ECL uses dynamic binary code generation to deploy specified computations into stones, supporting an anywhere, anytime code deployment model. Finally, for complex computations, the option exists to associate pre-compiled shared object modules with stones. Jointly, these features constitute a powerful means for dynamically extending the methods used for device access and control. We next explain how these are used to create custom camera devices in mobile settings.

4.3.1 CustomCam Architecture

Extending the basic V4L API, CustomCam provides an `ioctl` interface to the video driver for managing custom functionality, adding the request codes VIDIOSCUSTOM and VIDIOGCUSTOM to the existing set of V4L codes. These commands are used to set and get the custom functions associated with CustomCam, respectively. The prototypes of these calls

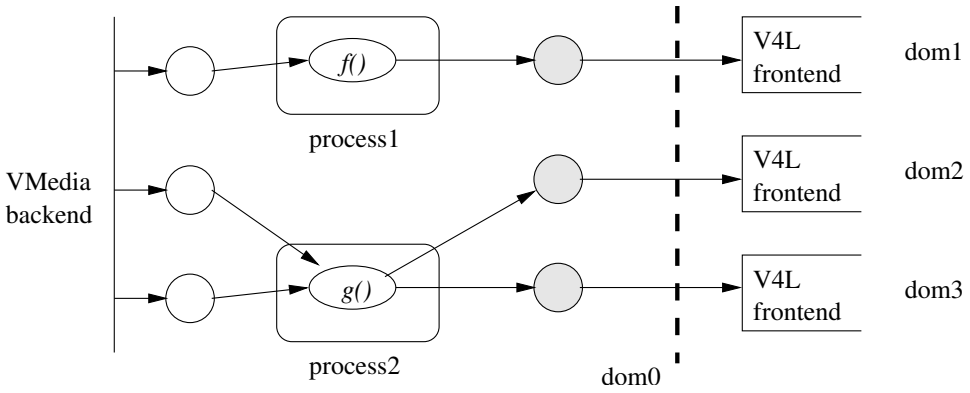


Figure 26: CustomCam Example Usage

are as follows:

```
int fd;
char *func;
int ioctl(fd, VIDIOSCUSTOM, func); // set
int ioctl(fd, VIDIOGCUSTOM, func); // get
```

A custom code, represented as an ECL function, is passed as a string argument. The first call sets the custom function to the string passed as `func`, whereas the second call obtains the current custom code. An example ECL function is shown in Section 4.3.4.

The overall architecture of CustomCam is illustrated with an example in Figure 26. Implemented as an extension of VMedia, CustomCam is shown serving three domains, dom1-3, all using the custom functionality feature. Stones are represented as ovals/circles. On the receipt of a VIDIOSCUSTOM `ioctl` call, the backend creates a new process (process1 in the figure, for domain dom1), and creates a stone within the process, deploying the given custom function to it. Next, it also makes the connections between output from the VMedia backend, as well as to the domain-specific backend. The domain-specific backend is linked with the frontend using Xen's event channel mechanism. In the absence of the VIDIOSCUSTOM call, the output from the VMedia backend is directly sent to the frontend, without any overheads. A NULL function passed via the VIDIOSCUSTOM call undoes any previous custom functionality already set, by freeing the stones and terminating the corresponding process.

4.3.2 Code Isolation and Accounting

In order to conform with the goals of virtualization, namely, providing isolation between VMs, every custom function is executed in a separate address space. As a result, a VM injecting malicious code will end up corrupting only its own device accesses, and other VMs are not affected. This allows mutually untrusting VMs to share the same device, with the illusion of exclusivity maintained.

In addition to the isolation mechanism, guest VMs can also benefit from recovery functionality. To handle cases where a VM ends up corrupting its custom functionality because of malicious code, the parent process can periodically check the status of its children processes and create a new process to continue serving the VM. On the other hand, the designer can also decide to notify the affected VM via an error code, on the next operation performed, thus allowing the VM to take its own recovery actions. The error notification already supported by CustomCam can be easily extended to perform such recovery.

Although custom functionality is performed in separate address spaces, such function are always run within the service domain, or dom0 – the domain that handles all device I/O. We exploit this fact to implement performance isolation. Specifically, since dom0 implements the functionality on behalf of the corresponding VM, accounting is performed by having each child process in dom0 periodically track its resource usage, using the SystemV `getrusage()` call. It shares this information with the VMM via hypercalls. In Xen, the VMM, then, can use this information to track the resource usage (and limit it, if needed) in the driver domain by each guest VM. This information, in conjunction with the information the Xen scheduler already maintains about the guest VM (i.e., time spent in various runstates – running, runnable, blocked and offline) can then be used to ensure fair resource allocation across each guest VM and the associated dom0 resident device extension functionality required/used by the guest. An extension of this functionality can provide support for Quality of Service (QoS), by making the underlying VM scheduler aware of the QoS needs of individual VMs.

4.3.3 Sharing in CustomCam

A potential advantage of solutions like CustomCam is the ability to share code and code executions across multiple clients. In CustomCam, the results of certain custom code could be shared by several mutually trusting domains that require the same functionality. This is achieved by maintaining a list of the custom codes set by VMs. If a new VM requests setting a custom code, it compares this against the list, and if a match is found, the overlay's stones are appropriately linked in order to share the same resources. The current implementation uses a simple string-based comparison of the codes, but a more sophisticated matching mechanism based on formal methods, or comparing the intermediate representations of the codes, can be used to detect nonidentical codes with the same functionality. In the example shown in Figure 26, domains dom2 and dom3 share the custom code defined by *g()*.

Since isolation is absent, it is important that only mutually trusting domains share their resources. Furthermore, in such cases, the resources spent in executing the custom code are accounted for equally across all the domains sharing it. We have not yet considered more sophisticated accounting methods. A related problem addressed by CustomCam is code reuse, where custom codes previously used by a domain are not discarded but saved for future use [152]. In response to new domains requesting the same custom code, the existing resources are simply reused, thereby speeding up code deployment.

4.3.4 CustomCam Usage Scenarios

All ECL functions in EVPath accept an `input` and an `output` as their parameters, and return an integer value. This return value determines whether the data should be further processed or dropped (depending on whether it is non-zero or zero, respectively). In CustomCam, this can be used, for instance, when an application requires images conditionally. An example is an application monitoring an area of interest that may be lit or not. If the application desires images from the camera only if the area is lit, it may select a custom code that performs this check and returns the image only if the check is successful. This functionality is particularly useful when the camera is remote because unnecessary frames discarded 'at the source' do not needlessly use potentially scarce network resources. Further,

though ECL codes allow some state to be maintained, it is primarily used for event-based computation, and hence, no state is assumed. This is especially important with VM migration. CustomCam maintains the state of the device entirely in the frontend, and hence state-maintenance during migration is handled by the VMM.

CustomCam makes it easy to use specialized hardware. For example, in the presence of an on-board DSP processor, some image processing functions specified via ECL can be executed directly on the specialized processor. In mobile domains, this can also be realized by offloading this functionality to a nearby, more powerful processor. Allowing the CustomCam implementation to make such choices based on currently available hardware frees the individual guest domains and their applications from adapting to platform specifics, including during VM migration.

The concepts embedded in CustomCam can be extended to other sensor devices. Since applications using sensor data typically perform continuous data acquisition and processing, they can potentially benefit from use-specific device customizability.

4.4 Implementation Details

The VMedia runtime is implemented as a user space application in the Service VM (Dom0) which completely encapsulates the I/O virtualization for the multimedia devices. Backend entities communicate with the frontends in guest VMs via Xen VMM-specific communication mechanisms, which provide for the shared message queues and signaling. Different physical device access entities are run as separate threads to provide maximize concurrency in the runtime. These threads use device-specific methods for I/O. For example, for a USB based camera, the corresponding thread uses V4L ioctl calls for image capture, similar to applications such as camE [3]. For a cellphone based camera, the corresponding threads communicates with a server process running on the cellphone that provides images over network.

For information dissemination between these edge nodes of the graph and to implement VMediaMD entities, the runtime utilizes an event-driven middleware, called EVPath [4]. EVPath allows data flow as *events* among nodes of an overlay termed *stones*. Stones can

perform event processing, and can can transform an input event to an output event, possibly of different type, before passing it on to another stone. Stones can also perform routing decisions based on the event contents. This allows EVPath to perform content adaptation, which is required to support the logical functionality provide by VMedia.

The VMedia framework allows two types of logical functionality – one encoded in the V4L attributes of the virtual device itself, e.g. image size and colormap, the other completely based on guest VM. In the former case, the VMedia runtime installs well-defined processing entities as stones in the MediaGraph. For example, if the image size of a virtual device is smaller than the physical device, a stone containing a scaledown filter is installed in the MediaGraph. Other filters, such as crop and grayscale, are installed in a similar fashion. VMedia also allows further predefined logical functionality via the extension of V4L attributes. For example, a virtual camera device may provide image data in specific image formats, such as JPEG and PNG. These functionalities can be provided in a manner similar to the earlier ones. These image processing-specific functionalities are implemented using the Imlib [5] library.

Current VMedia implementation as a user level application is one of the many possible ways to implement services in a virtualized environment. Another approach would be to implement this service at the kernel level in the dom0. It is also possible to build specialized domains for providing this service [18]. Although a detailed comparison of various approaches to implement VMedia functionality is out of the scope of this dissertation, we argue that a user level implementation provides certain advantages over other approaches. Specifically, current implementation allows for better debugging and ready reuse of many other software components, while providing reasonable performance tradeoffs. By utilizing shared memory communication channels, most of the data movement happens without any involvement of the dom0 kernel and Xen. Only time the dom0 kernel must be involved is to route signals between guest VMs and VMedia runtime. Since such signals are sent on a V4L message granularity (e.g., after the complete image has been transferred to the guest VM in response to a read), we anticipate that the overhead imposed by the user-kernel transitions in dom0 for VMedia runtime has minimal effect on the overall VMedia performance.

4.5 Experimental Evaluation

We evaluate the VMedia framework on a desktop system with 3.2GHz dual-core Pentium-D processor and 3GB of RAM. To this machine are attached a Kensington SE401 USB-based camera, and a second Motorola e680 cellphone with a built-in camera. The e680 cellphone runs the Linux 2.4.20 kernel and is connected to the desktop via USB. The camera communicates with the desktop using the TCP/IP protocol, supported by a virtual network driver over USB stack. Service VM (Dom0) running VMedia runtime is allocated 512 MB RAM and one of the physical CPUs, and runs the Linux 2.6.16 kernel. The other CPU is shared among guest VMs, as determined by Xen's scheduling policy. The VMM virtualizing the desktop system is Xen version 3.0.3.

4.5.1 Overheads of VMedia Framework

This set of experiments quantifies the overhead of multimedia virtualization via the VMedia framework, measured as the difference between the latency of image capture experienced by a guest VM from the virtual multimedia device and the latency of image capture experienced by the VMedia framework from the physical device. This overhead includes the cost of transformations performed on the media data (computation), and its dissemination to virtual device frontends (communication). The content is delivered only to those virtual devices that request it, even if these virtual devices share some (or all) of the VMediaMD components with other devices that did not request it. For these experiments, the image properties (size, palette etc.) for virtual and physical media devices are kept the same, so the only overhead incurred is due to dissemination.

The scalability of VMedia is demonstrated by increasing the number of VMs and measuring the *amortized overhead*. As the number of VMs is increased, transmission costs of VMedia runtime increase as media data needs to be disseminated to more and more VMs. However, the latency of image capture as experienced by a guest VM depends on the amount of sharing, as the cost of a physical I/O gets amortized over multiple virtual I/O requests. To capture this sharing effect, we only account for the net positive overhead experienced by a guest VM, which includes VMedia's dissemination cost. We average over all net positive

overheads experienced by N VMs sharing a physical device, and report it as amortized overhead. The overall cost of virtualization also increases due to scheduling, since context switching of VMs is required on a single CPU. In future multi- and many-core systems, the scheduling costs will be smaller, or even negligible, if there are enough physical CPUs.

We compare the VMedia overhead with a *time-sharing* approach of virtualizing the multimedia device. In this approach, every guest VM image capture request results in a image capture from physical device. In the presence of no contention, this approach is comparable to VMedia. However, in case when multiple VMs require access to the media device, the overhead of this approach not only includes communication of media data from the service VM, it may also include image capture latency from physical device for another VM. The overhead of this approach, hence, is always positive, and we report the average overhead per request.

We evaluate both VMedia and time-sharing approaches in two scenarios. In one scenario, termed 'no-wait', VMs successively request image capture from virtual devices without any wait between them. In another scenario, termed 'random-wait', a VM waits a random amount of time between $[0, 1]$ seconds before making another request.

Figure 27 compares the overheads of VMedia and time-sharing approaches. For a single VM, the overhead of both the approaches are negligible. However, as the number of VMs increase, the overhead of time-sharing approach increases rapidly, including multiples of physical capture time as a factor, in both 'no-wait' and 'random-wait', with latter being slightly better than the former. The overhead of VMedia approach also increases, but only due to the communication cost of VMedia and context-switching cost of VMs. Both of these overheads are small when compared to the physical capture time. The overall overhead of I/O for an image capture from the virtual device with increasing number of guest VMs becomes as high as $\sim 25\%$ of the overall virtual device capture cost.

For each scenario, we also present the *sharing factor*, which demonstrates the underlying approach's ability to share the device, the higher the better. This factor is calculated as $\frac{\sum_{i=1}^{N} \text{captures from virtual device i}}{\text{captures from physical device}}$, N being the number of guest VMs. Figure 28 compares the sharing factor for different virtualization approaches in two scenarios, as mentioned

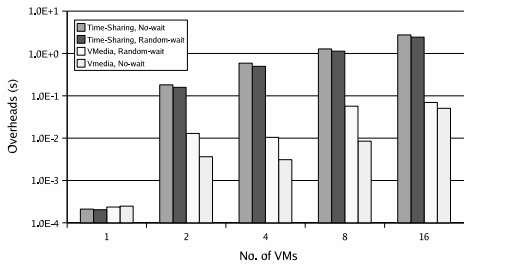


Figure 27: Overheads (y-axis is log-scaled)

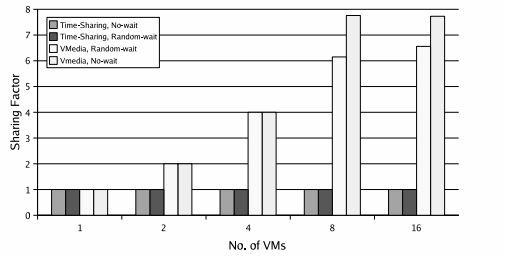


Figure 28: Sharing Factor

earlier. For perfect sharing, the sharing factor should increase linearly with increasing number of guest VMs. However, due to high context switching costs, we observe the best case sharing factor to be ~ 8. The sharing factor of time-sharing approach is always 1, since every virtual capture request results in a physical capture request. The VMedia approach attains best sharing factor for the 'no-wait' case, while the sharing factor reduces as the contention for the physical device is reduced in the 'random-wait' case.

These results show that the VMedia framework shares physical devices efficiently, which in turn contributes to its performance and scalability. Further, using higher level 'V4L' requests, the no-sharing passthrough type virtualization for a single VM can be achieved at a lower cost than, e.g., using USB level requests [84] – where every single USB level request adds an overhead of about 25%.

4.5.2 Enhanced Functionality Sharing

Results in the previous section demonstrate the performance benefits derived from device sharing and the consequent amortization of I/O costs. However, using MediaGraph, the VMedia framework affords further benefits by sharing at the *logical* level. To demonstrate the benefits of enhanced sharing via the MediaGraph, we construct the following scenario.

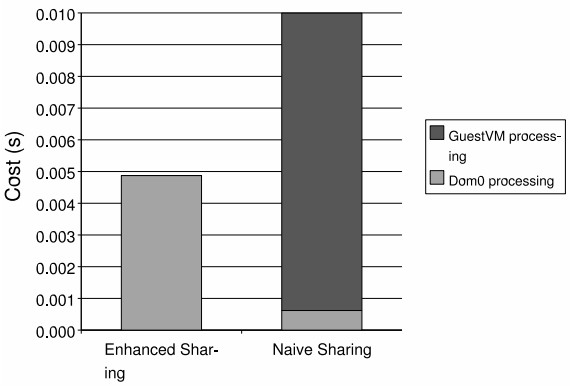


Figure 29: Enhanced and Naive Sharing

Four guest VMs are created in Xen – two VMs, VM1 and VM2, require images of size 640x480, while VM3 and VM4 require images of size 160x120. VM4 also requires grayscale images. We compare two approaches to sharing – the naive way, where the VMedia framework only shares the physical device and any transformations are performed by the guest VMs themselves and the enhanced way, where the VMedia framework also performs any required transformations. These transformations are derived by the framework based on the *parameters* of the virtual devices, namely image size and color palette.

In this case, since the MediaGraph reduces the amount of redundant transformations performed, we expect to see lower processing costs. As shown in Figure 29, since all transformation-related processing is performed in the Service VM with the MediaGraph, we see a higher cost. In the naive sharing case, the images are simply sent to all VMs. However, the guest VMs perform all of the transformations in the latter, which is completely absent in the former. The overall costs, as shown in the figure, are almost 50% lower due to elimination of redundant computation.

4.5.3 Dynamic Restructuring of MediaGraph

In this section, we quantify the overheads of VMedia framework associated with dynamic restructuring of MediaGraph. Restructuring is performed in response to the management actions performed on the virtual media devices. These actions include opening and closing of devices, and changing their properties, such as image size and color palette, via ioctl calls. The framework translates these actions into MediaGraph modifications, as described

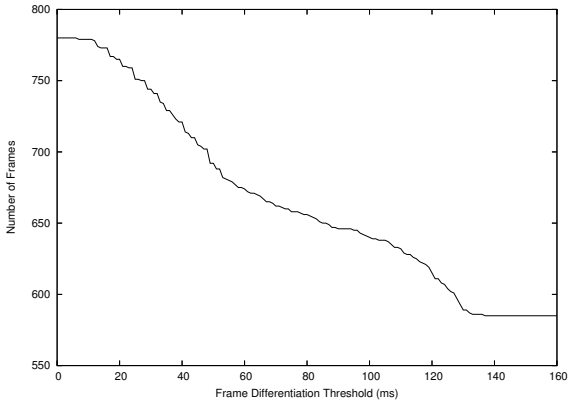


Figure 30: Number of Distinct Frames from Two Cameras in Response to Changing Frame Differentiation Threshold.

earlier in Section 4.2.3. The modifications require creation and removal of nodes from the graph, which in turn require EVpath *stones* to be added/removed.

We measure the cost associated with a management action as (1) the time it takes VMedia runtime running in dom0 to carry out the modifications, and (2) the amount of change in the MediaGraph resulting from these modifications. Since the removal of stones takes significantly less time compared to their additions, we only consider the number of stone additions as the metric for the amount of change in the MediaGraph. Figure 31 depicts these results. On x-axis, we vary the number of VMs (and hence the number of virtual devices). Each VM performs 100 management actions related to device property changes. Each action is drawn randomly-uniformly from a set of 5 such changes - 3 related to image size changes and 2 related to color palette changes. Each VM also waits for a random amount between $[0, 1000)$ milliseconds, between two consecutive actions.

Results demonstrate that with increasing number of VMs, the average cost per action decreases, since the cost of MediaGraph change could be amortized over actions from different VMs. Also, the amount of change required for MediaGraph increases sub-linearly. Put differently, the amount of change per management action decreases with increasing number of VMs. This explains the decreasing average cost of management actions.

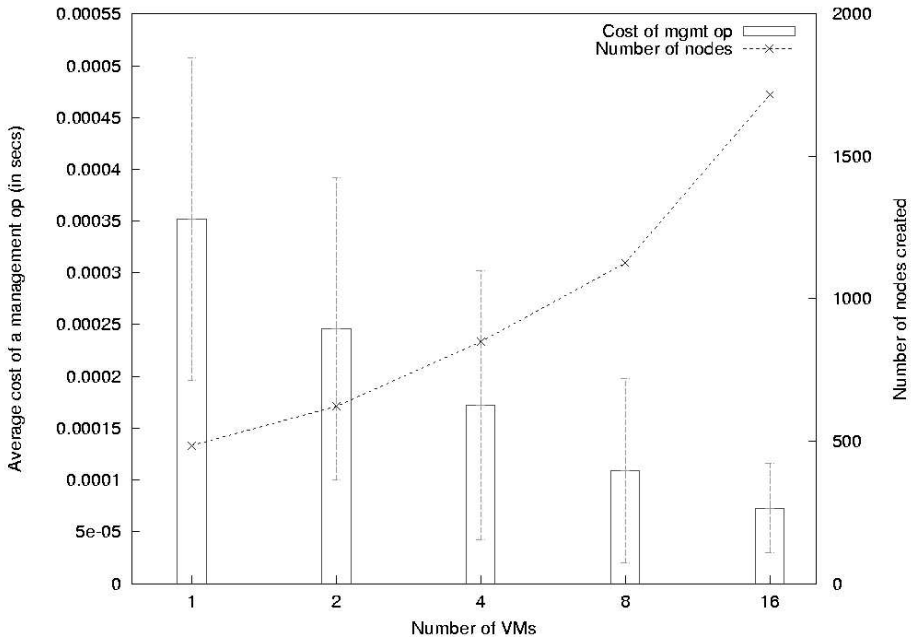


Figure 31: Management Cost of VMedia Runtime

4.5.4 Enhanced Logical Devices via Multi-Device Aggregation

Depending on the requirements of a guest VM and the availability of physical devices, certain services can be composed that allow a guest VM *improved* quality of service. For example, if a guest requests a wide image (of aspect other than regular 4:3), VMedia can aggregate images from multiple cameras either horizontally and/or vertically. Similarly, if a better resolution image is required, e.g. 640X480, but physical cameras can only provide 320X240, four such cameras can be aggregated. This is better than just using scaling – since it will not result in any quality loss. This can be further extended with additional processing to create a video wall [153]. Table 6 shows the microbenchmarks for a virtual camera device created by the concatenation of two cameras, the USB camera and the cellphone camera. The cost of physical device capture is taken as the maximum of these two devices, which corresponds to the cellphone camera. VMedia framework overhead includes the concatenation transformation action and the communication cost, and is very small compared to I/O latency.

Another example of aggregation is to use multiple media capture devices, possibly with a phase lag, in order to minimize the average latency of media access to guest VMs, where

Table 6: Cost Components for Multi-Camera Aggregation via Concatenation

	Cost (ms)	Cost (% of Vdev Cost)
Vdev Capture	622.023	100
Phys Capture	619.624	99.61
Transformation	0.907	0.15
Communication	0.366	0.06
Miscellaneous	1.127	0.18

these devices are sampling the same environment. For example, for a single continuous image source with interframe latency L, average latency for capturing an image is $L/2$ – assuming accesses arrive randomly over a uniform distribution. However, by using two such image sources, and running then with phase lag $L/2$, the average latency can be reduced to $L/4$, effectively doubling the frame rate. To demonstrate the viability of this approach, we use two cameras, one USB and one cellphone camera, to capture frames in parallel, in a time period T, and timestamp them. The latency of frame capture from USB camera is $\sim 200ms$, while from cellphone camera, it is $\sim 600ms$ ($\sim 300ms$ of which is the core physical capture latency on cellphone and rest of it is the network transfer over USB to desktop machine.) Next, we coalesce i'th frame from device 1 ($f_{i,1}$) and j'th frame from device 2 ($f_{j,2}$), iff $|T_{f_{i,1}} - T_{f_{j,2}}| < \delta$, where δ is the frame differentiation threshold. This threshold quantifies the difference in media content, and hence the value, provided by successive frames captured by different devices. At lower values of frame differentiation threshold, the added value of extra frame is less. The resultant number of frames denote the *valuable* content. We plot the resultant number of frames obtained in a 2 minute time-period against different values of δ, as shown in Figure 30. The result shows that the aggregate device can achieve more distinct frames than a single camera, and hence can provide better frame rate to the clients. Note that for high values of δ, the number of distinct frames are small, and asymptotically reach the number of frames provided by the faster device (~ 580 in this case), thereby limiting the benefits from using multiple devices. For lower values of δ, the number of frames are larger, although the difference in media content may be smaller, again limiting the benefits from using multiple devices.

Alternatively, such services can be created in the guest VM itself – if we provide one

Table 7: Costs of CustomCam `ioctl` Operations

Operations	Cost (ms)
`VIDIOSCUSTOM`	16.11
Process creation and setup	11.96
EVPath operations	4.11
Miscellaneous	0.04
`VIDIOSCUSTOM` (Sharing)	0.12
`VIDIOGCUSTOM`	0.03
Undo `VIDIOSCUSTOM`	1.16

virtual media device per physical device. This can be accomplished, e.g., by the VMedia framework itself, by creating multiple MediaGraphs, one for each physical camera. The passthrough access to physical devices can be utilized in a similar fashion. As argued earlier, the latter approach does not provide sharing, and hence is of little interest. We believe that a single MediaGraph with support for aggregation is better than aggregation in guest VMs, for the following reasons:

- The guest VM implementation couples virtual devices with the physical environment, e.g., in number of devices and their orientation. This is usually a concern in virtualized environments, since a VM may be migrated to a different physical platform. Hence, the service implementation on guest VMs must be able to adapt to any changes in the physical platform. This adds complexity for VMs. By keeping this functionality purely in the VMedia framework, the framework – local to a single physical platform – provides a better way to provide this service.

- A single MediaGraph allows for enhanced sharing, in case aggregation is utilized by multiple VMs. Computations for logical functionality can be performed once, and results can be shared among multiple guest VMs.

4.5.5 CustomCam Evaluation

We conduct microbenchmarks on the `ioctl()` operations supported by CustomCam. This consists of a `VIDIOSCUSTOM`, `VIDIOGCUSTOM` and a `NULL` call that undoes the operations of a `VIDIOSCUSTOM`. The results are shown in Table 7.

As shown, process creation takes up roughly 75% of the overall cost of setting up the

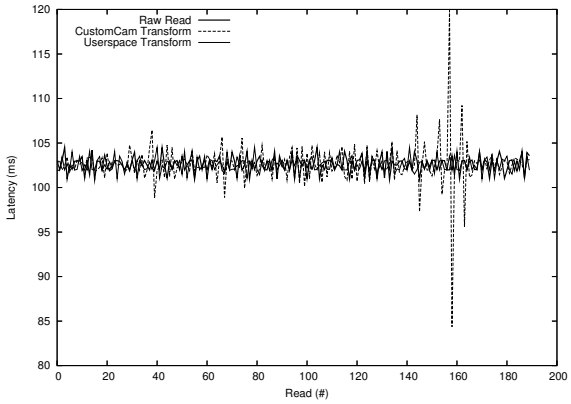


Figure 32: Read Latencies for Different Schemes

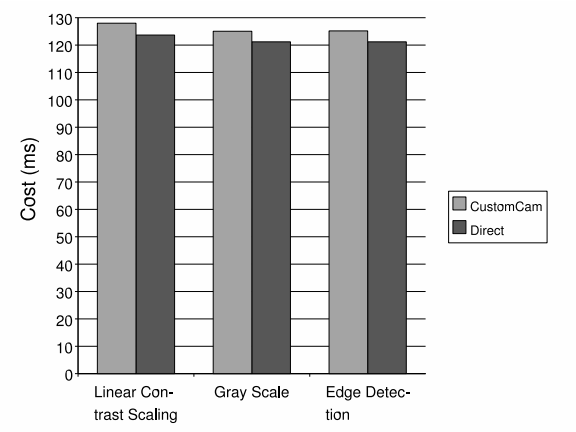


Figure 33: CustomCam vs. User-space Transformation

custom code, and EVPath operations take up most of the rest, with an overall cost of about 16.11ms. Since only the first such call needs to perform these actions, with all subsequent calls simply sharing the same process and stones, subsequent calls require only 0.12ms. Undoing the `VIDIOSCUSTOM` call, i.e., freeing the process and the stones, costs roughly 1.16ms, whereas getting the custom code is a very light-weight operation, since it involves only a send to the appropriate domain.

Next, we apply example custom codes and compare them against executing the respective code within the domain. Figure 33 shows these costs for three image transformation codes drawn from real world applications – grey scaling, linear contrast scaling and edge detection. On average, the performance attained using CustomCam is very close to that when transformation is performed within a domain, in user space. It is to be noted that

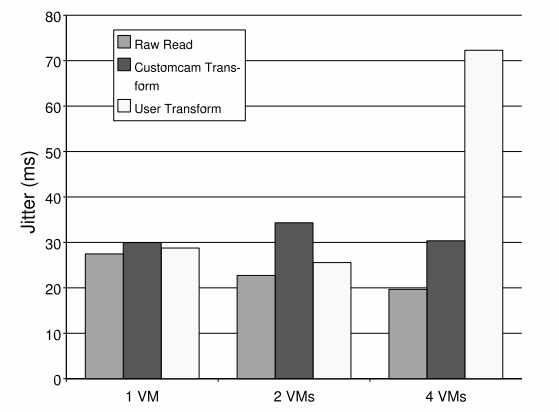


Figure 34: Jitter Comparison

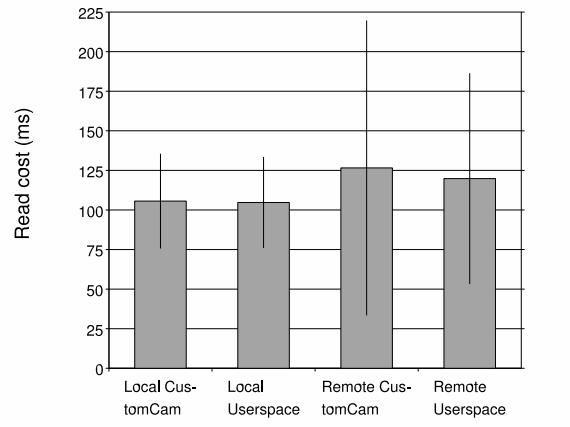


Figure 35: Remote vs. Local Access

these ECL codes were hand optimized due to the absence of many optimizations in binary code generation. Since the binary code generator is designed to be invoked during runtime and has a limited memory footprint, its output is unable to match compiler generated code's optimizations. This can be rectified by enabling the execution of functions in shared objects. The loss in portability of such a scheme is traded off against the gain in code performance.

Figure 32 plots the latency observed for each read access, over multiple reads performed continuously. Three cases – (i) reading the image, (ii) reading the image when transformation is handled by the CustomCam, and (iii) reading the image with transformation handled by the user space – are compared. The figure shows that with CustomCam, the jitter is mostly comparable to and only slightly higher than for the other two cases. Since the

transformation action runs in the user space of a different domain(dom0) in CustomCam, scheduling actions on both the domains affect the latency of its reads. However, Figure 34 shows the effect of number of VMs on the jitter observed. Grey scaling is used as the transformation, and with CustomCam, this code is shared among the VMs. Jitter is measured as the standard deviation of read latencies. As the number of VMs is increased, the jitter experienced by CustomCam and raw reads stays almost constant, whereas with user space transformation, the value increases rapidly. These benefits are achieved due to sharing: as CustomCam allows different VMs to perform their transformations in a centralized manner, repeated switching among VMs is reduced.

The final portion of the experiments are conducted across two instances of CustomCam, running in two separate nodes linked by a Gigabit ethernet switch. A domain served by CustomCam makes use of a remote camera, which is physically accessed by another instance of CustomCam. Reads are performed on the devices repeatedly and the latencies measured. As seen in Figure 35, remote reads incur slightly higher latency and jitter than local ones. The remote CustomCam implementation has a slightly higher cost and jitter than performing a remote read and carrying out the transformation itself locally, in user-space. However, if the transformation can also perform filtering – i.e., selectively allowing remote images depending on their content, unnecessary reads can be avoided. CustomCam allows this functionality whereas the user-space transformation does not.

4.6 Discussion

VMedia and CustomCam demonstrate the role that middleware could play in presenting device enhancements to VMs. The implementation details of the devices can be hidden from the VM effectively using such techniques. However, depending on the implementation, middleware failures, if not appropriately handled or exposed to the VM, may present problems due to the VM's assumption of the existence of physical devices. It is also noted that overuse of these functionalities may lead to increased dependence of the VM on other domains' capabilities.

However, this separation also presents several benefits including those discussed in the

chapter. A key benefit is to allow existing software including the operating systems, to work with rapidly improving hardware functionalities. Recent trends indicate the penetration of multi-cores in everyday computing. Coupled with a sensor-rich environment that future mobile platforms are expected to work in, techniques outlined in this chapter permit these hardware improvements to occur independent of slower adaptation of existing operating systems code. An extension of this concept to include services, as shown in the next chapter, provides more opportunities to decouple software and hardware units.

CHAPTER V

VIRTUAL COOPERATIVE PLATFORMS WITH V(IRTUALIZED) SERVICES

Chapter 4 addressed the sharing of local/remote devices among virtual machines, using middleware-based services. In this chapter, we extend this theme, to include arbitrary services to be offered, used and shared in this manner, and offering several capabilities to the virtual machines, found useful in the mobile/pervasive environments. We also show how arbitrary middleware can serve as the backend for such an implementation.

In this chapter, we address the problem of creating consistent and uniform execution environments for guest virtual machines and applications in mobile systems. The approach, termed *Virtualized Services* (VServices), enhances the execution platforms provided by hypervisors (or Virtual Machine Monitors, VMMs) with additional mechanisms and methods to virtualize the *services* required by guest VMs. Since service virtualization is performed at a higher level of abstraction than the devices virtualized by typical VMMs, opportunities are created to introduce optimizations to improve resource utilization and access and/or to share services so as to remove redundancies in their use. In addition, new or enhanced devices and device capabilities can be supported, without introducing additional complexities into guest VMs. Similarly, service realizations can be based on physical devices, on software enhancements of such devices, or on software device emulations. Finally, remote service access, even with mobility, can be implemented with any or multiple of the many existing middleware-based access methods, without requiring guests to adopt specific middleware solutions (e.g., .NET vs. Java).

Basic properties of VServices demonstrated in this dissertation include the following:

- low overhead of service access and activation,
- high performance in service execution, and

- opportunities for service consolidation leading to improved efficiency in resource use

5.1 Design of Virtualized Services

Services afford the clean separation of an implementation from its use, and are a popular option for separating levels of abstraction in software. They may exist within a single machine, such as system calls, or across machines, like remote procedure calls. They may also differ in their interaction types (request-response, publish-subscribe, etc.) or their interfaces (object-based, web-based, etc.). This generality in services lends them applicable to a variety of scenarios. As a result, several middleware techniques rely on service-based architectures [112] to interact with applications.

In current virtualization mechanisms, devices are virtualized using hosted virtual machine architecture [139] (or a similar mechanism), where one domain (termed the 'host') is permitted direct access to the physical devices (using the corresponding device drivers belonging to the OS running in the domain). I/O from other domains are directed by the hypervisor to the host, which mediates access to the device. Paravirtualization techniques use a similar mechanism – for instance, Xen uses a split driver model where device I/O of guest domains are handled by the frontend device driver within the domain, which interacts with a backend in the control domain via the hypervisor's sharing infrastructure. The backend uses native device drivers to directly access the hardware.

In order to support this mechanism, modern virtualization techniques support methods to share memory among VMs, in addition to low latency message passing between VMs. Device virtualization is then built on top of this infrastructure by using the memory sharing mechanism to transfer data and the messaging to emulate interrupts. For instance, Xen supports the event channel interface coupled with shared pages for these purposes [118].

5.1.1 Virtualized Services Architecture

Virtualized services reuse the same mechanisms used by VMMs for device virtualization. Shared memory is used to transfer data between the service provider and the consumer, and the messaging infrastructure is used for control (such as initialization, parameter changes, shutdown, etc.). Although these mechanisms do not impose any restrictions on the service

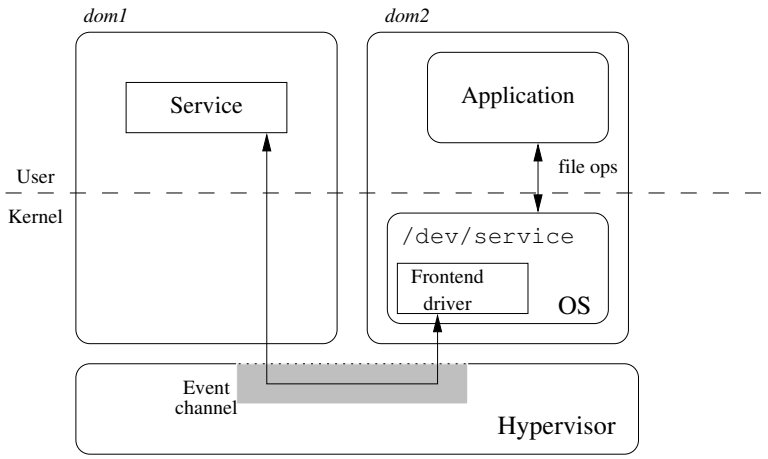


Figure 36: Virtualized Services Architecture

interfaces and hence allow general services to be implemented, we reuse the file-based interfaces commonly used in all UNIX-based systems for the following reasons: (i) this interface is standardized so that adapting it to non-UNIX platforms is also straightforward, (ii) it makes bundling services with existing devices simpler, and (iii) since operating systems have been optimized around this interface, its efficiency does not pose a concern.

The overall architecture of VServices, as implemented in the Xen VMM hosting two Linux domains is shown in Figure 36. The domain *dom1* acts as a service provider, and an application in domain *dom2* uses this service. The OS inside *dom2* provides a device frontend which exports a `/dev/service` file as a character device (termed service device). Any application needing to use the service accesses the service device via regular file operations, which are translated to the appropriate commands by the frontend device driver for the service, which in turn are exchanged with *dom1* via the event channel interface provided by the hypervisor.

In the remainder of the section, we discuss the design and implementation of two types of services – directory, and group communication services, both realized using the VServices architecture.

5.1.2 Directory Service

A directory service uses a request-response based interaction to perform directory lookups based on request parameters, and serve the information retrieved back to the requester.

Table 8: File operations and service semantics

File operations	Directory	Pub-Sub
`open`	init	init
`close`	cleanup	cleanup
`write`	send request	publish event
`read`	receive response	receive event
`select/poll`	wait for response	wait for event
`ioctl`	control operations	channel management

Common directory services include Domain Name Service (DNS), and LDAP-based address book or yellow pages lookups. A virtualized directory service implementation propagates writes to the service device into the domain that hosts the service (or acts as a proxy to the service). Here, the write is translated to a request call. The response returned by the service is communicated to the requesting domain via a soft IRQ. Upon a read, this response is passed on to the application. Additionally, ioctl() calls exported by the service device are used to set and get parameters (server name, buffer sizes, etc.). Errors from the service are transparently reported as appropriate file access errors. Due to the generic nature of the interface, this service can be extended to work as RPC service (Sun-RPC, XML-RPC) or web services (SOAP-based) with light modifications.

5.1.3 Group Communication Service

The group communication service is realized using an event-based publish-subscribe middleware. Pub-sub middleware provides transparent construction and maintenance of multicast trees among participating nodes in a group, and exposes simple interfaces to enable communications (via publish and subscribe). In the virtualized group communication service implementation, writes from the application are translated to publish calls, and received events are passed to the application when read calls are issued. As in the directory service, ioctl() calls are used to set and get parameters (buffer sizes, for example) as well as to initiate control actions (creation of a channel, subscribing to a channel, etc.). Table 8 summarizes the interfaces used by these services.

VServices are accessed by applications via the service device, and hence require applications to use the device interface instead of service calls. In order to allow legacy applications

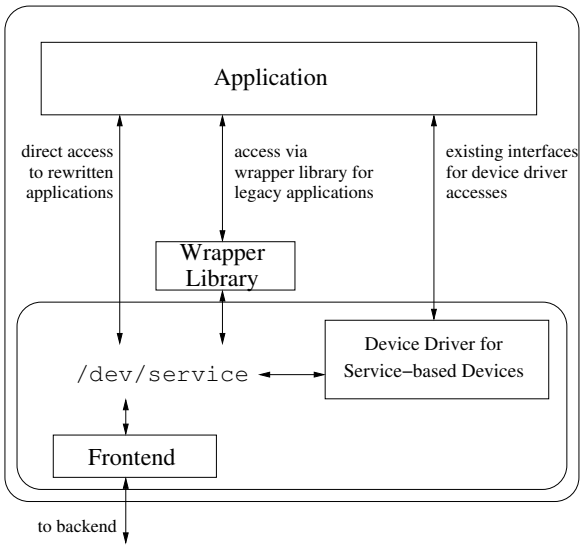


Figure 37: VService - Application Interactions

to use this interface, a library is used to provide wrapper calls that translate service calls to device accesses (Figure 37). Applications using the service via dynamic libraries require no changes to use VServices. However, those that are built via static linking need to be rebuilt to link with the wrapper library.

The entire state associated with each VService is stored in the guest VM's frontend (in addition to in the backend), and hence requires no changes to the implementation of VM migration. On the commencement of migration, the backend simply discards its copy of the state corresponding to the VM, and the state is communicated to the backend on completion of a migration, which is then handled by the new backend. For instance, the channel identifier of a pub-sub channel is stored in the frontend as well as in the backend, and on migration, the new backend acquires the ID from the frontend and completes subscription, while the old backend unsubscribes from the channel and discards the channel ID.

Although VService has been implemented and demonstrated using network-based services, it must be noted that it is not limited to such services. For instance, other capabilities such as vector processing, cryptographic operations, etc. can be offered as local services via the VService interface.

5.2 Virtualized Services Management

As VServices provide centralized implementations of service functionality, they present opportunities to improve service management, either via performance optimizations or quality management, discussed next.

5.2.1 Performance Optimizations

The two primary capabilities enabled by the centralized service implementations in VServices are

- Concurrency: As many service implementations within individual domains are avoided and the corresponding service access stack eliminated (for instance, a domain running a SOAP client application can get rid of its own implementations of HTTP and the entire network stack), this leads to lightweight OSes in guest domains with smaller memory footprints. The consolidation of the implementations also leads to more efficient scaling as compared to the isolated case. Additionally, any specialized processor/other hardware in the platform that can speed up service implementations can be utilized effectively, especially if a guest OS does not possess capabilities to handle the hardware.

- Sharing: VServices enable the semantic information exchanged upon their use, to expose opportunities to share functionality. For instance, consider a virtualized DNS service where DNS requests and responses from guest domains are relayed to the nameserver, by the backend at the control domain. If the backend now caches the responses for the various requests locally, the responses to previously unrelated requests from separate domains can possibly be sped up. On the other hand, sharing also opens up potential security/privacy concerns. In the same example, a domain receiving an immediate response to the first DNS request it makes could infer that some other domain had recently made the same request.

5.2.2 Quality Management

Another benefit of semantically higher-level virtualization is the opportunity to specify and implement meaningful Service Level Agreements (SLAs). For instance, instead of guaranteeing network packet level metrics agnostic of the data movement, content-based service level guarantees can be provided. These can also be enhanced with policies for sharing services among domains, to address security related issues that arise as discussed in Section 5.2.1. Further, the ability to decouple the quality of service implementation for a domain from its scheduling priority is also enabled, giving greater flexibility.

5.2.3 Limitations

VServices provide benefits as outlined in Sections 5.2.1 and 5.2.2, besides enabling the innovative functionality discussed in Section 5.3. On the other hand, VServices virtualize at levels of abstraction – at the application level – for which there may not exist well-defined, standard APIs. This can be remedied by emulating the standard file- or object-based interfaces used in operating systems or by exploiting subsystem-specific standards (e.g., using the Linux v4L interface [124] or the T10 standard for object-based file systems [120]. Another remedy is to use wrapper libraries OS virtualization, of providing an abstraction that is closest to the bare hardware [151]. An issue with all such solutions, however, is that during VM migration, additional support is required for dealing with the VServices being used. Since standard hypervisors do not provide such support, this results in the need for extensions in the 'host' domain.

5.3 Device Enhancements

Reusing the split driver model designed for devices to implement VServices provides several means for transparently provisioning these services along with an existing device driver interface. Examples include extending device functionality, device emulation (discussed in Sections 5.3.1 and 5.3.2), device remoting (linking a local device driver with a remote device), device consolidation (using the functionalities of different devices to present a single emulated device), device functionality isolation, etc.

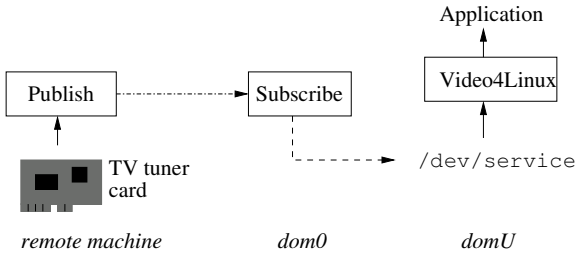


Figure 38: VService-based TV Tuner

5.3.1 Extending Device Functionality

Extending an existing device's functionality is useful in some scenarios. An example of this is providing a uniform set of functionalities to guest VMs, independent of the physical device's capabilities. Any deficiencies in the capabilities are made up for using services that emulate these features. For instance, Linux uses the common Video4Linux(v4l) device driver API to access multimedia devices such as camera, TV/radio tuners, encoder/decoders, etc. Here it is possible to provide a TV/radio tuner functionality to the VM even though the physical device may not support it. The multimedia streaming content for the tuner device can be provided by a service. In this work, we use the capabilities of a USB-based webcamera to provide a v4l interface to the guest VMs, but enhance it to provide TV tuner capabilities as well, and rely on the service backend to subscribe to a channel in a publish-subscribe middleware for the media content. Figure 38 shows the design of such a device.

This mechanism can be further extended to provide capabilities that do not exist in *any* physical devices at all. For instance, content from a location-based information service can be overlaid on content from the camera to provide location-based images. An example of such a functionality is achieved using a location- and orientation-based service that provides information such as the directions to nearest conveniences overlaid on the image from the camera itself. An extension of this notion allows users to compose a chain of services coupled with devices to provide rich platforms to the guest VMs, resembling the system architecture for pervasive computing project described in [62], using simple techniques similar to web service mashups [73], already used in the OS context [69].

5.3.2 Device Emulation

Emulating a non-existent device to the guest VM is one of the many features provided by modern virtualization frameworks, and the split driver model designed in Xen enables this using simple implementations. In contrast to the traditional approach of realizing device emulation using software at the backend [124], or by linking to a remote device [84], here we rely on services. Additionally, we also gain the flexibility to switch between services, depending on a diversity of factors, so that the device implementation can also be variable. In this work, we describe how a Global Positioning System (GPS) device, that provides location information based on timing differences of signals received from various GPS satellites, is emulated indoors using services. We use two types of methods here: (i) using the strength of signals from various bluetooth devices in the vicinity and the predetermined knowledge of these signals at various locations in the area [68], and (ii) detecting the position using external cameras that detect and track the target, then deduce its position based on a precalibrated scale. Each of these are available as services, and the backend can choose the service to be used based on factors such as the environment (i.e., whether the target is in the camera's range), accuracy desired (between the two methods), power consumed (bluetooth vs. camera), etc. By not possessing the physical device itself, or in situations when it cannot be used (such as indoors), other alternatives are made available using this technique. Figure 39 shows the design of such a device, where the shaded boxes represent remote components and the clear boxes, local ones. Here, we assume that although the bluetooth signal strengths are determined by the local device, an external service that translates signal strengths to precise coordinates (based on previous surveys) exists as a network-based service. Current GPS devices are typically accessed via the serial port, and vary slightly in their interface to applications. As a result, higher level daemons have been developed (e.g., gpsd in Linux) to present a standard interface to applications. VService-based localization can be emulated as a serial device, or allow applications to access via a modified gpsd wrapper daemon.

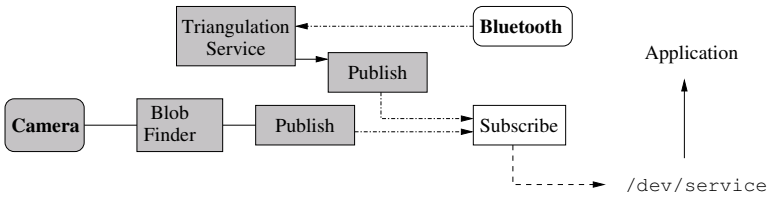


Figure 39: VService-based Indoor Localization

5.4 Experimental Evaluation

VServices is implemented in Xen 3.0.4 running a Linux kernel 2.6.16.33. Experiments are conducted on a Dell Latitude notebook with a 1.66GHz Intel Centrino Duo dual-core processor, and 1GB memory. The control domain, or dom0, uses both the cores for execution, and 512MB memory. All guest VMs share one of the two cores, with each VM getting 64MB of memory, permitting a maximum of 8 VMs to be run concurrently. A directory service and a publish-subscribe service are implemented. The directory service is implemented using the domain name service as an example. It translates `write()`'s of server names to IP addresses, which are then accessed using `read()` calls. The publish-subscribe service uses the `ioctl()` call for channel management and buffer sizing, as discussed in Section 5.1.3. The backend implements the ECho pub-sub middleware [48], a portable event-based middleware which provides support for installing event filters at run-time using dynamically generated code. However, the interface is general enough to allow substitution of other event-based pub-sub middleware in its place. Time measurements use the `rdtsc` instruction for accuracy.

5.4.1 Service Costs and Scalability

Our first set of experiments measure the costs involved in performing DNS service calls, with the hostname of another host in the same LAN, as the request. As Table 9 shows, the main contribution is from the `write()` call, which signals the backend about the new request, and blocks on the response. The backend directs the request to one of the threads in a thread pool, which performs a `gethostbyname()` call, that involves searching the local cache (`/etc/hosts` in Linux) for a match, and if found to be missing, obtaining the IP from the nameserver. The `init()` and `connect()` calls are involved in opening and connecting a socket to the nameserver, while `send()` and `recvfrom()` perform the lookup, following

Table 9: Cost breakdown for the DNS VService, starred operations occur in backend

Operation	Cost (cycles)
`open`	4232
`write`	736691
∗socket `init`	30760
∗`connect`	5660
∗`send`	14247
∗`recvfrom`	564064
∗socket `close`	11154
`read`	7175
`close`	35543

which the socket is closed. The blocking `recvfrom()` call takes up about 90% of the overall backend costs. `read()` performed at the guest VM is a simple copy from the buffer. `close()` releases the thread back to the pool.

Figure 40 compares the overall cost of performing the DNS call using the VService implementation, against the traditional `gethostbyname()` implementation from the guest VM. `gethostbyname()` takes roughly 1ms for each lookup, when only one VM is present, and shows a small increase for up to 4 VMs, but shoots up rapidly to over 7ms for 8 VMs. Since each VM performs these operations over its own TCP stack and relies on the control domain only for access to the physical device, there is considerably more work. In contrast, the VService implementation scales much better since most of the work is performed by the control domain. Even for a single domain, VService exhibits about 50% lower latency. The benefits are entirely due to low overheads, since no caching of IP is undertaken in the control domain (i.e., each DNS request is propagated to the nameserver in all the cases).

Next, we evaluate VServices for high performance, using the pub-sub implementation. In these experiments, another host in the same network creates a channel, and applications inside the VMs publish or subscribe to the channel. During each run, 1000 events of size 4KB are sent continuously by the publisher(s), and several runs are conducted. Figures 41(a) and 41(b) show the performance of publish and subscribe operations, as the number of VMs (and hence the number of publishers/ subscribers) is increased. It compares the naive implementation that uses the guest VM's virtual network, against use of VService. In the case of publish, we notice that the naive implementation performs better

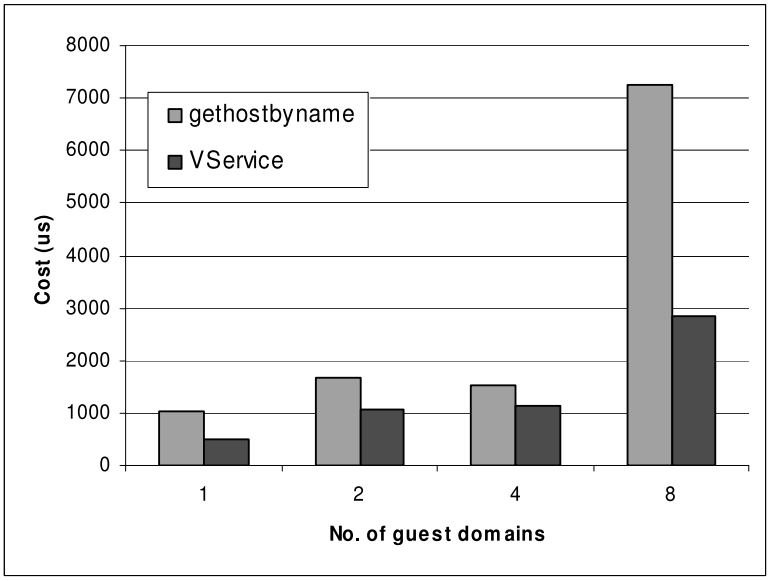
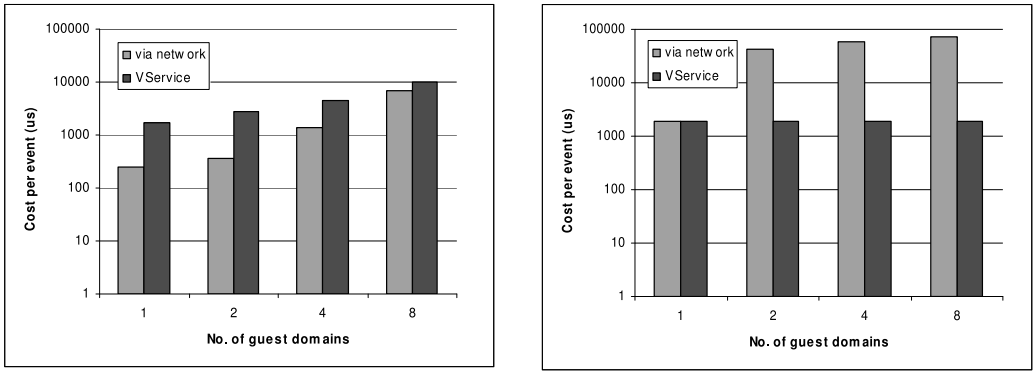


Figure 40: DNS Costs Comparison and Scalability

than VService, but scales worse. The primary reason for this is that publish operations translate to blocking `write()` calls on the device, which are handled only in batches by the backend (in order to minimize frequent VM context switches). This can be rectified by having a thread continuously handle these operations at the backend, without waiting for explicit signals from the frontend. While this may improve performance, it may also increase the CPU utilization. Alternatively, by changing the buffer size, it is possible to obtain better throughput (discussed in Section 5.4.2).

In the subscribe scenario, we find almost equivalent performance with the naive- and VService-based subscribes for the single VM case, but the latter scales much better than the former–in fact, it almost stays constant, as the number of VMs increases. The reason for this behavior stems from two factors – (i) the implementation of the ECho middleware, and (ii) the VService `read()` implementation. Firstly, since each subscriber in the naive implementation has a unique IP address, a multicast tree is constructed by ECho such that each VM becomes an endpoint. The same events traverse through multiple virtual interfaces to reach the endpoints. In the case of VService, there exists only one such endpoint, and the events are copied to the buffers corresponding to each VM, which are then directly



(a) Publish

(b) Subscribe

Figure 41: Costs Comparison and Scalability for Publish and Subscribe

read by the applications. Since these copies are inexpensive, they do not significantly add to the overall cost. Secondly, subscribe buffers are implemented as ring buffers, where the application can read from the buffer concurrently with backend's `write`s to it. As a result, the throughput is not affected as long as the CPU is not fully-utilized.

5.4.2 Services Management

With the use of multiple, or specialized cores, allowing the backend to efficiently manage the core assignment to various operations becomes possible, as discussed in Section 5.2. In this experiment, we evaluate the assignment of the VService backend operations to the same core as the VM, or to the other core, and report our findings in Table 10. While the DNS example has only one running VM, the pub-sub example has two VMs (no gains were found with only one VM). However, the gains are found to be very small (less than 5% in all the cases). We may attribute this to the fact that since VService is implemented using blocking `write()`'s, the service consumer is idle during service fulfillment as well, and this dependency limits parallelism. In the case of subscribe, we find that the backend is able to fill the subscribe buffers with incoming events much faster than the frontend can read and pass them on to the application. This stems from the batched nature of event handling (in order to minimize VM context switches), that allows for limited gains. We may conclude that, to realize the potential of multiple cores, a redesign of the current VServices

Table 10: Effect of core use

VService	Cost (us)	
	same core	different core
DNS (1000 requests)	412.4	407.1
Publish (per event)	3013.4	2898.4
Subscribe (per event)	1902.0	1902.6

implementation to allow for non-blocked writes and unbatched reads is necessary.

VServices are implemented with a total buffer size of 64KB, though this can be changed easily. The read and write buffer sizes can be changed to values less than this limit, implying that the frontend will signal the backend for handling data as soon as the limit is reached, and vice-versa. As `write()`s are currently implemented as blocking writes, and `read()`'s buffers are implemented as ring buffers, this leads to differences in their behavior with respect to buffer size changes. Since each event is 4KB in size, writes would block for each event when the buffer size is 4KB or less, whereas with a 32KB buffer, every eighth write is blocked before it is handled, along with the previous seven writes. This allows latency to be traded off for throughput. `read()`s do not follow this behavior and show steady performance, for reasons previously mentioned in Section 5.4.1. Figure 42 shows the results, which closely follow our discussion. Note that these effects do not apply to the DNS service, since latency is important in DNS interactions. Consequently, *any* write/read is handled immediately by the other end.

5.4.3 Device Enhancements

To implement the TV tuner VService example, we use another Linux desktop in the same network with a 3GHz dual-core Pentium 4 processor and 512MB memory, that possesses a Conexant Brooktree 878 PCI-based TV tuner card to act as the media server, connected through a 100Mbps switch. We use the fftv-0.8.3 open source software to grab frames from the card using the V4L driver, and publish it using ECho (Figure 38). VService is used to subscribe to this channel and obtain the frames inside the guest VM. The three different strategies we evaluate in this experiment are: (i) Local frame grab using the `read()` system call, (ii) Local frame grab using the `mmap()` system call, and (iii) Remote frame grab using

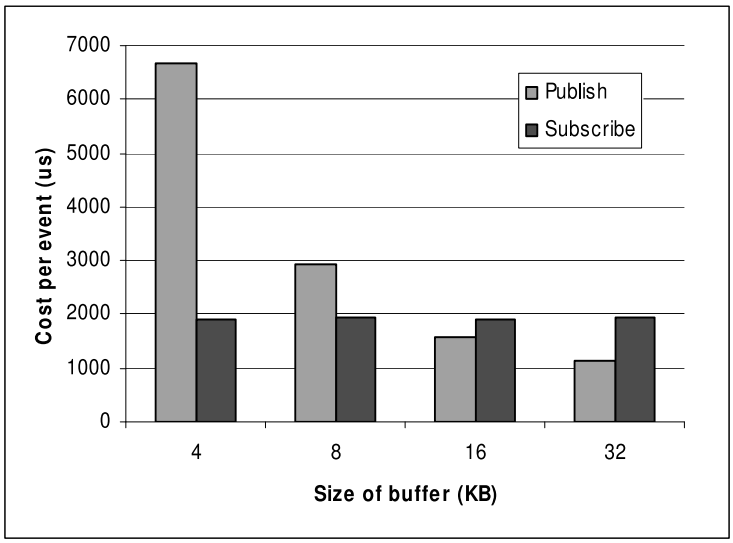


Figure 42: Effects of VService Buffer Sizes on Pub-sub Costs

VService, that uses the `mmap()` based call to access the TV tuner. Frames are continuously grabbed and the period between frames noted. We also evaluate three sizes of images – 160x128 pixels (large), 96x64 (medium), and 48x32 (small). These sizes are chosen in order to avoid the need for compression. The same software (fftv-0.8.3) is used within guest VMs to obtain the frames from the driver to the application. The results are shown in Figure 43, along with the jitter values represented using error bars. VService adds a 50% overhead to `mmap()` based calls while transferring the frames across the network, and higher jitter values to smaller frames. It is noted that these costs are comparable to local `read()` based frame grabs.

For the indoor localization device, we present costs for each of the operations involved in obtaining the data. We use a Logitech Quickcam USB-based webcam with the gspca-based V4L driver to capture images, then process them using the CMVision blobfinder tool, which detects objects with pre-defined colors in the image as "blobs". The position of the blob is then translated to absolute physical coordinates, using precalibrated readings (the camera is assumed to be static). These coordinates are published through an ECho channel. The bluetooth-based localization is performed using a USB-based Belkin bluetooth adapter attached to the local machine. We assume the presence of other fixed bluetooth devices in

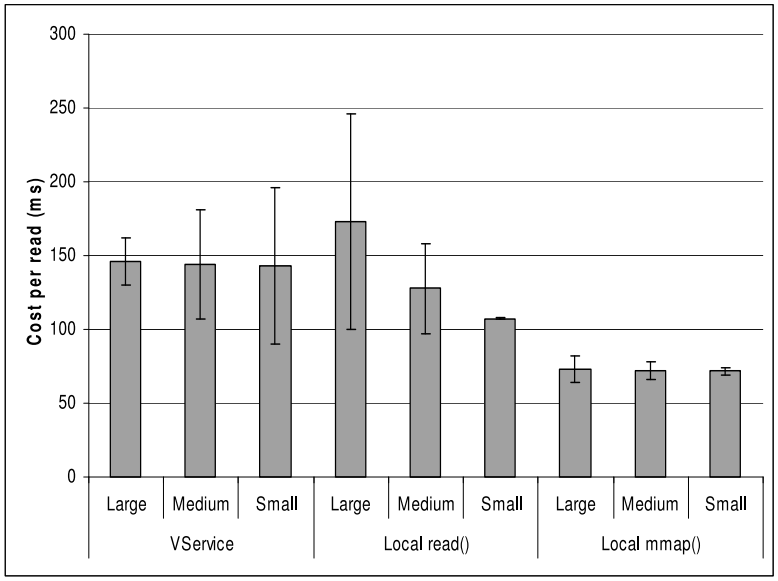


Figure 43: Cost per Frame Using Different Grab Techniques, on Different Frame Sizes

Table 11: GPS device component costs

Component	Cost (us)
Frame grab	108.33
Blob detection	4210.18
Bluetooth scan (s)	11.85
Bluetooth connect (s)	6.47
Bluetooth link quality	5095.40
Pub-sub delay	213.25

the vicinity that act as beacons, as mentioned in Section 5.3.2. We present the basic costs of this service in Table 11. The time taken to grab a frame, process it, and receive the coordinates add up to less than 5ms. The time taken to determine link quality and infer coordinates might be slightly higher (depending on the number of beacons, as well as link quality to coordinate conversions), however, this can still be performed well under 1s, the latency that outdoor GPS devices typically offer. Despite the low latency in determining the link quality of a bluetooth device, the time involved to scan for other devices and connect to them are significantly higher. Fortunately, these are one-time operations, and do not introduce additional latencies after a connection is established.

5.5 Discussion

VServices have been implemented in Xen, and uses its split driver model. However, as mentioned in Sections 5.1 and 6.6, other virtualization techniques use mechanisms similar to the split driver model to implement device virtualization. Hence, VServices can be ported to other virtualization mechanisms with little effort. Similarly, as VServices use the standard device interface, and since the device drivers used in conjunction with VServices provide the actual interface with the applications that use it, adapting it to work with various operating systems takes the same effort as porting a device driver from one operating system to another. We also note that the shifting of functionality from one VM to another (usually the control domain) enabled by VServices does not pose any scalability issues, since (i) VServices do not increase the overall computation, but merely shift some processing from one VM to another, and (ii) the number of VM context switches does not increase due to VService. In fact, often it is the opposite, due to the use of higher level functionality. As noted towards the end of Chapter 4, middleware in conjunction with virtualization provides several opportunities to decouple software and hardware entities. For future mobile and pervasive systems, this decoupling enables the rapid composition and use of specific applications, from basic ubiquitous software and hardware components. Extending the ideas of dynamically composable platforms, providing application users with the tools to compose their own applications has the potential to introduce new paradigms to pervasive computing [154].

CHAPTER VI

RELATED WORK

6.1 Middleware Systems

Middleware systems (like CORBA [149], DCOM [32] and Enterprise Java Beans [127]) enable the construction and execution of componentized software across distributed systems. A main focus of such systems is interoperability across heterogeneous hardware and operating system platforms. Middleware has also been used in the high performance domain. PM-2 [143] permits MPI-based applications to run over a cluster with heterogeneous network interfaces. Grid computing [53] has enabled parallel computing on the Internet. The middleware listed above has evolved from the client-server domain, typically using RPC or related communication paradigms across communicating components. Other useful execution models include messaging [60, 78], publish-subscribe [28, 46], and distributed query models [162].

A common paradigm used by implementations of modern distributed middleware is that of overlay networks (an early use of which is described in [65]). Sometimes termed 'underlays' when used with higher level models like data streaming or publish-subscribe, the use of overlays provides degrees of freedom in how nodes and node communications are managed beyond those provided by underlying operating systems, networks, or hardware. They may provide robustness in naming or resource location for peer-to-peer applications [138], they may be enhanced with autonomic management methods and algorithms [85], and enable adaptation to changes in network behavior [35] or provide resilience to failures [19]. MSO uses overlays to realize its component deployment and reconfiguration services.

6.1.1 Middleware for Mobile Systems

Middleware for MANETs has experimented both with overlay-based approaches and with alternative programming and execution models. Event-based publish-subscribe systems have also been extended recently to the mobile systems domain [70, 39]. Adaptability includes

opportunistic overlays [38] in which overlays are adjusted to cope with runtime changes in network topology, including runtime service redeployment. A similar approach is used by Solar [37], which allows context discovery using application-specified operators deployed over the network for pervasive systems. Expeerience [30] is a JXTA-based middleware that has been adapted to run on MANETs. MagnetOS [90] uses a distributed framework to partition a monolithic Java application into its constituent classes for cooperative execution on a MANET. Efficient placement of application components is carried out by monitoring data traffic among nodes, and by using distributed algorithms to shorten the mean path lengths of data transfers. Unlike these middleware solutions which use Java, MSO is written in C/C++ for more fine grained monitoring, accounting and energy management. DFuse [83] manages distributed fusion of data streams, performing dynamic assignment of fusion operators to nodes and adapting it at run-time, including for iPAQ handheld devices. MSO supports fusion among other operators, and is also able to provide other features such as heterogeneity and adaptation to mobility, absent in DFuse.

Smart messages [79] leverages prior work in Active Networks [144] for MANET systems. This middleware permits specialized distributed programs, termed smart messages, to autonomously execute at nodes of interest, and route themselves towards their destination via migration. This is further extended in [125] so that migration is driven by the operating context. While this is suitable for delay tolerant lightweight applications, it is not suitable for the cooperative applications targeted by MSO. XMIDDLE [98] is a middleware framework for reconciling multiple inconsistent replicas of a file (arising from changes made when disconnected) on a multi-hop ad hoc network. It represents XML documents as trees and implements protocols to reconcile differences in trees. The COMMAND [25] middleware provides transparent rebinding for client-server interactions through the use of proxies elected in an ad-hoc fashion. Proxies are responsible for dealing with dynamics in the environment. COMMAND also utilizes code mobility to move functionality to proxies on-demand, thus enabling the proxies to provide multiple services. This removes the necessity to have all of the functionality residing at all nodes. [26] discusses extending the mobile-agent based middleware paradigm to mobile computing. A survey of various

middleware implementations for mobile environments appears in [97].

6.1.2 Robot and Sensor Networks

In the area of autonomous robotics, Miro [146] is a CORBA-based middleware designed to support multi-platform robotics. It also provides generic services like localization which can be applied to different platforms. Another CORBA-based middleware is presented in [155]. These middleware do not provide adaptation to mobility and energy management, the important features in MSO. Middleware solutions for sensor networks include MiLAN [66], which collects quality of service details from the application layer and network conditions from the sensors, then decides on the best configuration to support data aggregation under current constraints. Impala [92], the middleware used in Zebranet, is designed to support adaptability and allows easy software updates of sensor nodes. MSO is not optimized for sensor networks, as it supports not only data aggregation, but other data flow models as well.

Self configuration is an important theme of all of the work presented above. [119] identifies the design paradigms for self-configuring networks to be (i) maintaining local behavior to achieve global properties, (ii) enabling simple coordination, (iii) minimizing long-lived state and (iv) adapting to changes. The MSO approach attempts to address these requirements by (i) providing local remapping, (ii) making simplifying assumptions during remapping and conservatively resorting to global remapping if these are violated, (iii) maintaining state locally, and (iv) monitoring for and reacting to changes.

Unlike the middleware approaches described in this section, MSO takes a holistic approach to run applications in the MANET environment by recognizing that several different higher level mechanisms in such systems can be built from basic low level abstractions and services. This allows autonomic management mechanisms to be more easily constructed/extended from these services.

6.2 Energy Management in Mobile Systems

Prior research on energy conservation in ad-hoc networks has mainly focused on energy-aware routing protocols – by improving existing protocols like AODV [115], by factoring

the energy levels of each node in the routing cost metric [64], or through novel protocols like probabilistic routing [134]. The former classifies nodes into various classes depending on their energies, and uses this instead of the hop count, to determine the energy-optimal route. The latter uses a similar cost metric by aggregating the energy values of the nodes in each route, then randomly chooses a route with a probability proportional to the cost metric along the route. While these approaches are suitable for network-bound applications and sensor nodes, where the network interface accounts for a significant portion of the power budget, for the applications and platforms considered in our work experimental results that the energy consumed by the network interface is quite small compared to CPU energy. Our research, therefore, primarily leverages prior work on reducing CPU energy consumption, including reducing energy usage by applying dynamic voltage and frequency scaling [75] on multiprocessor systems. [94] uses dynamic slack reclamation in conjunction with DVFS in a real-time setting, on a multiprocessor system. A static schedule is first constructed for periodic tasks, then slack reclaiming is used to save power, yet satisfy real-time constraints. Resource reclaiming in multiprocessor real-time systems has been dealt with in great detail in [135], where the authors describe two algorithms to perform online reclaiming on a static schedule. The impact of user interfaces on energy saving schemes is studied in detail in [166]. Coolspots [114] exploits the varying power consumption profiles of various wireless radios, and switches to the appropriate radio, based on the desired level of performance. Through this scheme, the authors demonstrate up to 50% savings in energy.

Computational offloading has been used extensively for power-aware load balancing ranging from clusters of workstations [117], to embedded devices [159]. The latter performs computational offloading in conjunction with setting the CPU frequencies, to minimize energy consumption. Distributed middleware can be useful to manage computational entities in such environments. [97] surveys the various middleware implementations for a mobile environment. Research efforts to include mobile nodes in Grid technology include [33], which proposes a mobile agent framework to provide/use Grid services at the mobile nodes, so that distributed resources from the Grid can be accessed by such users.

6.3 Device Virtualization

Efficient methods to virtualize basic system resources like CPU and memory have been well studied. Recent efforts in virtualization have focused on efficient sharing of I/O devices such as network interface [123]. The VMGL [86] approach virtualizes a video card to provide hardware-based 3D acceleration to guest VMs. As identified in the paper, standardized higher level interfaces improve both the ease of implementation and the adoption of such solutions. VMGL uses the OpenGL abstraction as the interface, whereas the VMedia framework uses the Video4Linux [13] interface. The Boxwood project [96] of the Singularity operating system provides storage at higher abstraction levels such as B-Trees for example, hiding lower level disk virtualization details from applications. The T10 industry-standard for next generation SCSI devices [12] is another interface that is amenable to device virtualization, particulary for object stores.

Device virtualization is dealt with in different virtualization techniques in different ways. Xen and VMware use the methods outlined in Section 5.1. KVM [21] uses similar methods, but relies on Qemu [51] running on the host kernel for device emulation. User mode Linux [74] relies on the parent kernel for device access, since its kernel runs as a user-space process. In order to improve device virtualization, providing exclusive unmediated control of a device by a single VM has been studied. The PCI passthrough [163] feature in Xen allows such access to the PCI device by a single VM, and thus avoids control domain overheads, but it also precludes sharing. Efforts to improve the network device virtualization in Xen VMM include Xensocket [164], which recommends bypassing the TCP stack and using copy instead of page flipping, when exchanging data between domains. This shows significant improvements in throughput, especially with large message sizes. VServices similarly avoids the guest VM TCP stack for network accesses, allowing the backend to package the data on the guests' behalf. Other research has proposed several network optimizations to Xen, that include performing hardware-based optimizations either at the hardware (if they support) or in the control domain (if hardware doesn't support these optimizations) [100]. The VService approach allows such an optimization to be cleanly implemented for services by completely bypassing the virtual network interface.

6.4 Middleware-based Device Enhancements

Aggregating multiple devices to provide richer services has been studied along several dimensions. Superimposed projection [42] discusses fundamental issues arising when using multiple projectors to produce a single high-resolution image. The Princeton scalable display wall project [153] also discusses algorithms to solve alignment, color balancing, and other problems arising in a distributed environment. The Lyra system [160] studies timing services that can be provided to multimedia applications, for achieving better quality of service. Such services can be harnessed in multimedia scheduling in a virtualized environment to provide QoS guarantees to guest VMs, as well as to schedule fine-grained captures in frame aggregation (Section 4.5.4) for example.

Research focusing on sharing multimedia include the Irisnet project [59], which applies filtering on distributed multimedia sensors to deliver customized content. Feeds from several remote webcams connected to the Internet are used to compose useful content and services built on top. MSODA [157] proposes a multimedia service overlay among VMs for media service access and composition. VMedia framework focuses on providing multimedia services to VMs via higher level 'logical' devices, while services are implemented in the Service VM. The Indiva middleware [108] also provides a higher level, file system abstraction, for composing distributed multimedia content. The Ninja project [61] builds abstractions to simplify scalable application construction over wide area networks. MSO does not explicitly support multimedia workload, but multimedia applications and query algorithms can be implemented using the services provided by it.

6.5 Customizable Devices

The extension of data streams with application-specific codes has been studied at multiple levels of abstraction. At the network level, in Active Networks [152], code injected by a user to a router operates on and possibly modifies, packets passing through it. Further, Smart Messages [80] enable user-defined distributed applications to execute over a wireless network of embedded systems. At middleware levels, runtime code injection is a key part of publish-subscribe software (e.g., ECho [48]), which uses this to filter and modify events,

based on a subscriber's requirements. CustomCam exploits the methods used in ECho to implement this, namely, via ECL codes [47]. Though ECL does not guarantee type-safety, other type-safe implementations (e.g., Cyclone [77]) could also be used.

Safety properties must be guaranteed when applications are permitted to extend trusted domains. Proof carrying code [105] is one way to attain safety, whereby the code carries a 'proof' generated by a certifying compiler, that the code satisfies the properties an agreed safety policy. The proof can then be verified, prior to execution. Safe languages are another way to provide safety properties [27]. In contrast, practical techniques like those used by kernel plugins [57] exploit certain features of computer architectures to guarantee isolation between extensions and OS kernels. Our work uses a combination of language safety and hardware support to isolate the extensions performed by different applications and to continue to guarantee safe operation when extensions are faulty or malicious. The Nooks project [140] extends this idea to isolate device drivers into lightweight protection domains inside the kernel address space, and further restarts a failed driver transparent to the client. The idea of isolating device drivers into their own VMs has also been studied previously [88].

As discussed previously, the concepts discussed in this dissertation are implemented using the VMedia multimedia virtualization framework [124]. While VMedia addresses media devices, its concepts can also be used to virtualize arbitrary devices and services [122]. Finally, some of the concepts presented in this work have been evaluated for a kernel-level implementation, with the CameraCast [82] set of mechanisms, which enforce capability based differential data protection on remote multimedia feeds, exploiting the aforementioned kernel plugins [57] isolation mechanism.

The concept of extended or 'smart' devices has been shown useful in many other contexts, including with our own work on self-virtualizing network interfaces and on smart NICs [15], the latter performing application-specific processing of network packets at the device level, thereby freeing the CPU to perform other work [58]. 'Smart' devices have also been shown useful for storage [14].

6.6 Virtualized Services

Recent efforts in middleware have resulted in implementations based on the concepts of Service Oriented Architecture (SOA) [112], and service virtualization [158]. SOA eases the creation of enterprise applications by allowing the composition of various services that serve to accomplish specific tasks, to align with business processes. Service virtualization helps in adapting the SOA model to a heterogeneous environment, by providing consistent interfaces for its management. Combining middleware concepts with virtualization mechanisms, it benefits from the well-defined interfaces provided by SOA and from management mechanisms such as load balancing, migration, isolation and resource monitoring provided by virtualization. As a result, it is gaining adoption in grid computing [54]. VService combines services with virtualization techniques at a lower level, thus making device emulation and composition possible, by interfacing services with device drivers. Service Oriented Device Architecture (SODA) [44] allows accesses to devices via services (in contrast to the VService approach of presenting services via device interface), in order to simplify device management. It is developed for use in enterprise systems, where the same interfaces designed to access enterprise services are also used to access and control devices (such as RFID tags, and other sensors or actuators). UPnP [145] (Universal Plug and Play) is a middleware standard designed to allow linking of devices in a seamless manner. Targeting personal area networks such as a home network, it aims at minimizing user involvement in setup. VService can be easily adapted to such a setup, by replacing the directory or group communication services demonstrated in this paper with UPnP compatible protocols. The use of component middleware to implement resource-intensive applications such as software defined radios, radar systems, etc., using parallelization techniques is described in [128].

Use of the device interface for implementing services is a well-established practice, originating with Unix file-based APIs. Examples include the use of the `/dev/random` device in several flavors of UNIX to generate pseudo-random numbers in software. Other examples include the `/dev/evtchn` in Xen [118], [156] to exchange events between domains, and `/dev/binder` in Openbinder [110] to exchange data between components. The Plan 9 operating system [116] extended this idea to include all resources in the system to be available

via the file system interface. For instance, a TCP connection is made by accessing files under `/net/tcp`.

The Libra [17] library operating system extends the Exokernel idea [50] of providing customized operating system to applications, thus delivering only the functionality needed by the applications. Libra provides services required by a Java application running within a JVM, by implementing frequently accessed services locally and relying on the hypervisor for other services. Libra uses the 9P distributed file system protocol to access remote services, whereas VService offers common services implemented by the control domain, for use by other VMs. Both these efforts are valuable in minimizing the size of guest operating systems. Proxos [142] allows negotiation of trust between a commodity operating system and a running application so that parts of the application remain secure if the operating system's security is compromised. It allows this by the application, splitting its system calls into those that are handled by the commodity OS and those that would be handled by its own private OS. Liquid VM [24] runs a Java VM with middleware directly on a hypervisor, in order to reduce OS overheads.

There are multiple ways to virtualize devices. Xen and VMware use the methods outlined in Section 5.1. KVM [21] uses similar methods, but relies on Qemu [51] running on the host kernel for device emulation. User mode Linux [74] relies on the parent kernel for device access, since its kernel runs as a user-space process. In order to improve device virtualization, providing exclusive unmediated control of a device by a single VM has been studied. The PCI passthrough [163] feature in Xen allows such access to the PCI device by a single VM, and thus avoids control domain overheads, but it also precludes sharing. Efforts to improve the network device virtualization in Xen VMM include Xensocket [164], which recommends bypassing the TCP stack and using copy instead of page flipping, when exchanging data between domains. This shows significant improvements in throughput, especially with large message sizes. VServices similarly avoids the guest VM TCP stack for network accesses, allowing the backend to package the data on the guests' behalf. Other research has proposed several network optimizations to Xen, which include performing optimizations either with hardware (if supported) or in the control domain (if hardware doesn't

support these optimizations) [100]. The VService approach allows such an optimization to be cleanly implemented for services by completely bypassing the virtual network interface. Vmedia [124] and Netchannel [84] are examples of efforts aimed at extending device functionality and transparent use of remote devices in the VM context, respectively.

CHAPTER VII

CONCLUSIONS AND FUTURE WORK

7.1 Conclusions

In this thesis, we have presented novel approaches to address the challenges encountered in running applications over cooperative mobile platforms, and have also extended our solutions to virtual mobile platforms. We adopt a service-based approach, where existing middleware techniques are used to implement services that are used by the application and higher level algorithms to adapt to dynamics in the mobile environment. We demonstrate the practical benefits of our approach by evaluating its implementation on real-world applications.

We develop an overlay network based middleware to enable cooperating nodes in a mobile ad hoc network to collectively execute an application, using decentralized management techniques. We develop general monitoring and (re)configuration services that permit autonomic management of the platform. We demonstrate higher level functionality such as load balancing and energy management built from services provided by the middleware. Energy-aware reallocation provided by MSO could increase the lifetime of an example robotics workload running on a network of five prototype handheld devices, by about 14%. Slack reclamation capabilities provided by MSO was found to closely match optimal frequency and voltage settings in a similar network.

We identify device management and sharing in a virtualized system as a key problem in cooperative mobile platforms, and develop middleware-based techniques to share, emulate, enhance and customize device properties in such an environment, then demonstrate our techniques experimentally on a multimedia device. Features enabled by our low-overhead middleware include the ability to emulate a single multimedia device using a collection of devices, better scalability on the number of virtual machines sharing the device, and a roughly 50% savings in sharing costs. Implementation of device customization techniques

described in this dissertation show negligible overheads, and low jitter values for the multimedia device, in both local and remote settings.

We extend our ideas to share arbitrary services among virtual machines in a cooperative mobile platform, and implement middleware-based service sharing. The service latency is reduced by 50% using this method, and improved scalability for high-throughput applications were demonstrated. Additionally, practical realizations of novel devices were built using these services, to explain the usefulness of the technique.

In summary, the primary contributions of this thesis are:

- developing middleware-based services to build autonomic management functionality in a cooperative mobile platform,
- applying middleware-based services to virtual platforms, thus enabling enhanced device functionalities, and
- identifying higher-level features that can be built on top of these services, that are directly useful to applications.

7.2 Future Work

The ideas presented in this dissertation can be applied to a variety of mobile scenarios. In the personal computing space, methods to access each others' computing resources may lead to novel use of such functionality, in environments with diverse device presence such as a home, as well as in areas such as concerts, conferences, and public areas where a large number of people with mobile devices gather. In the latter case, there is also a potential for supporting social networking-based applications. In such cases, the mobile devices may be enhanced with devices made available via the surrounding infrastructure, one example being making display boards in airports and railway stations suitable for use by surrounding users (with the appropriate usage policies). As explored in Chapters 2 and 3, mobile robots possess several sensors, and the applications they run make extensive use of the sensor data. MSO can be used to support such application execution over a network of mobile nodes. Additionally, the use of one robot's sensors/actuators by another can

be enabled using ideas discussed in subsequent chapters. This capability is also useful in vehicular networks. Modern automobiles and roadside infrastructure carry several devices to sense their environment. Sharing of such devices can aid in better vehicular navigation (by informing the driver about accident hot spots, congested areas, etc.) and automate some aspects of driving. Advances in wireless technologies further serve to extend existing pervasive applications into the vehicular space. This thesis can be extended in three major areas.

Services developed as part of MSO can be used to provide other management functionalities not considered in the thesis, such as for instance, assigning application modules to nodes that are best suited to run them. The monitoring capabilities provided by MSO can be used to understand detailed requirements of the modules, and accordingly mapped to an appropriate node. Moreover, while the management techniques provided by MSO are satisficing in nature (since its goal is to arrive at a node mapping for the application as quickly as possible in a decentralized manner), it would be a useful step to consider if optimality can be achieved. For instance, mapping application components to underlying nodes with the goal of minimizing energy consumption or of guaranteeing minimal end-to-end delays. Solutions based on heuristics can serve as a good starting point for such efforts.

Secondly, the device enhancements and sharing features demonstrated in this thesis used webcamera as the example. More sophisticated cameras with onboard processing capabilities (e.g., Faymax cameras) being developed in the market allow some of the VMedia/CustomCam functionalities pushed onboard, rather than on the control domain as demonstrated in the thesis. Further, techniques developed to share multimedia data can be improved to share arbitrary data from other types of sensing devices.

Finally, the services developed with VServices demonstrated its use with network-based services as example, but there is no conceptual reason to restrict ourselves to these types alone. Other types, such as (i) cryptographic services (provided by IBM's crypto cards, for instance), (ii) vector-based services in a Cell-like multi-core processor or graphics accelerator, (iii) storage-based services such as provenance, etc. can also be used with VServices. In conjunction with virtual devices enabled by VMedia/CustomCam efforts, VServices can

be used to develop dynamic composable platforms, in which all the services and the devices available, along with the interfaces, are presented to the application designer/user, who decides how they can be combined to address a specific application. For instance, an advertising display board in a public spot can serve as a game console and images from surveillance cameras can be used to control the game, in addition to a user's handheld computer. Other surrounding devices may also be used to share the computational load as necessary. Several challenges need to be overcome before this vision is realized. Some of them include (i) maintaining directory services that enable users to query services/devices available for use, (ii) addressing isolation, sharing, accounting and QoS features in this scenario, (iii) handling and recovering from failures, and (iv) agreeing on common standards for data and control transfers.

REFERENCES

[1] "Android." http://code.google.com/android, accessed April 2008.

[2] "ARM11 MPCore." http://www.arm.com/products/CPUs/ARM11MPCoreMultiprocessor.html, accessed April 2008.

[3] "CamE." http://directory.fsf.org/camE.html, accessed April 2008.

[4] "EVPath." http://www.cc.gatech.edu/systems/projects/EVPath/, accessed April 2008.

[5] "imlib." http://freshmeat.net/projects/imlib/, accessed April 2008.

[6] "Internet small computer systems interface." RFC 3720.

[7] "Mobile linux." http://www.linux-foundation.org/en/Mobile_Linux, accessed April 2008.

[8] "Network file system version 4 protocol." RFC 3530.

[9] "OS X." http://www.apple.com/iphone/features/index.html#macosx, accessed April 2008.

[10] "QNX Neutrino." http://www.qnx.com/products/neutrino_rtos/, accessed April 2008.

[11] "Symbian." http://www.symbian.com, accessed April 2008.

[12] "T10." http://www.t10.org/.

[13] "Video4Linux Resources." http://www.exploits.org/v4l, accessed April 2008.

[14] Acharya, A., Uysal, M., and Saltz, J., "Active disks," in *Proceedings of the Conference on Architectural Support for Programming Languages and Operating Systems*, 1998.

[15] Adiletta, M., Rosenbluth, M., Bernstein, D., Wolrich, G., and Wilkinson, H., "The next generation of the intel ixp network processors," *Intel Technology Journal*, vol. 6, no. 3, 2002.

[16] Adwankar, S., "Mobile corba," in *Intl. Symposium on Distributed Objects and Applications*, 2001.

[17] Ammons, G., Appavoo, J., Butrico, M. A., Silva, D. D., Grove, D., Kawachiya, K., Krieger, O., Rosenburg, B. S., Hensbergen, E. V., and Wisniewski, R. W., "Libra: a library operating system for a jvm in a virtualized execution environment," in *Proc. of VEE*, 2007.

[18] Ammons, G., Appavoo, J., Butrico, M. A., Silva, D. D., Grove, D., Kawachiya, K., Krieger, O., Rosenburg, B. S., Hensbergen, E. V., and Wisniewski, R. W., "Libra: a library operating system for a jvm in a virtualized execution environment," in *Proc. of VEE*, 2007.

[19] Andersen, D., Balakrishnan, H., Kaashoek, F., and Morris, R., "Resilient overlay networks," in *ACM Symposium on Operating System Principles*, 2001.

[20] Arnold, K. and Gosling, J., *The Java Programming Language*. 1996.

[21] Avi Kivity and Yaniv Kamay and Dor Laor and Uri Lublin and Anthony Liguori, "kvm: the linux virtual machine monitor," in *Ottawa Linux Symposium*, 2007.

[22] Aydin, H., Melhem, R., Mosse, D., and Mejia-Alvarez, P., "Power-aware scheduling for periodic real-time tasks," *IEEE Transactions on Computers*, vol. 53, May 2004.

[23] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A., "Xen and the art of virtualization," in *Proceedings of the* 19^{th} *Symposium on Operating Systems Principles*, ACM Press, October 2003.

[24] BEA, "LiquidVM." `http://e-docs.bea.com/wls-ve/docs92-v11/config/lvmintro.html`.

[25] Bellavista, P., Corradi, A., and Magistretti, E., "Lightweight code mobility for proxy-based service rebinding in manet," in *International Symposium on Wireless Communication Systems*, 2004.

[26] Bellavista, P., Corradi, A., and Stefanelli, C., "Mobile agent middleware for mobile computing," *IEEE Computer*, vol. 34, pp. 73–81, 2001.

[27] Bershad, B. N., Savage, S., Pardyak, P., Sirer, E. G., Fiuczynski, M. E., Becker, D., Chambers, C., and Eggers, S., "Extensibility safety and performance in the SPIN operating system," in *Proceedings of the* 15^{th} *ACM Symposium on Operating Systems Principles*, pp. 267–283, ACM Press, 1995.

[28] Birman, K., "The process group approach to reliable distributed computing," *Communications of the ACM*, vol. 36(12), pp. 36–53, 1993.

[29] Birrell, A. D. and Nelson, B. J., "Implementing remote procedure calls," *ACM Transactions on Computer Systems*, vol. 2, no. 1, pp. 39–59, 1984.

[30] Bisignano, M., Calvagna, A., Modica, G., and Tomarchio, O., "Expeerience: A jxta middleware for mobile ad-hoc networks," in *Peer-to-Peer Computing*, 2003.

[31] Braun, T., Siegel, H., Beck, N., Boloni, L., Reuther, A., Theys, M., Yao, B., Freund, R., Maheswaran, M., Robertson, J., and Hensgen, D., "A comparison study of static mapping heuristics for a class of meta-tasks on heterogeneous computing systems," in *Hetereogeneous Computing Workshop*, 1999.

[32] Brown, N. and Kindel, C., "Distributed component object model protocol – dcom/1.0," in *Internet Draft, Network Working Group*, 1998.

[33] Bruneo, D., Scarpa, M., Zaia, A., and Puliafito, A., "Communication paradigms for mobile grid users," in *International Symposium on Cluster Computing and the Grid*, 2003.

[34] Bustamante, F. E., Eisenhauer, G., Widener, P., Schwan, K., and Pu, C., "Active streams: An approach to adaptive distributed systems," in *Proceedings of the 8^{th} Workshop on Hot Topics in Operating Systems*, 2001.

[35] Cai, Z., Eisenhauer, G., He, Q., Kumar, V., Schwan, K., and Wolf, M., "Iq-services: Network-aware middleware for interactive large-data applications," *IEEE Concurrency and Computation: Practice and Experience*, 2006.

[36] http://carmen.sourceforge.net/.

[37] Chen, G. and Kotz, D., "Solar: An open platform for context-aware mobile applications," in *International Conference on Pervasive Computing (short paper)*, 2002.

[38] Chen, Y. and Schwan, K., "Opportunistic overlays: Efficient content delivery in mobile ad hoc networks," in *International Middleware Conference*, 2005.

[39] Chen, Y., Schwan, K., and Zhou, D., "Opportunistic channels: Mobility-aware event delivery," in *Middleware*, 2003.

[40] Chou, Y., Fahs, B., and Abraham, S., "Microarchitecture optimizations for exploiting memory-level parallelism," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, June 2004.

[41] Clausen, T., Dearlove, C., and Dean, J., "Manet neighborhood discovery protocol." IETF work in progress draft, 2008.

[42] Damera-Venkata, N. and Chang, N. L., "Realizing super-resolution with superimposed projection," in *Proceedings of IEEE International Workshop on Projector-Camera Systems*, 2007.

[43] Das, A., Spletzer, J., Kumar, V., and Taylor, C., "Ad hoc networks for localization and control," in *IEEE Conf. on Decision and Control*, 2002.

[44] de Deugd, S., Carroll, R., Kelly, K. E., Millett, B., and Ricker, J., "Service oriented device architecture," *IEEE Pervasive Computing*, vol. 5, no. 3, 2006.

[45] Diamos, G. and Yalamanchili, S., "Harmony: An execution model and runtime for heterogeneous many core systems," in *HPDC*, 2008.

[46] Eisenhauer, G., Bustamante, F., and Schwan, K., "Event services for high performance computing," in *IEEE International Symposium on High Performance Distributed Computing*, 2000.

[47] Eisenhauer, G., "Dynamic code generation with the e-code language," Tech. Rep. GIT-CC-02-42, College of Computing, Georgia Institute of Technology, 2002.

[48] Eisenhauer, G., Bustamante, F. E., and Schwan, K., "Event services in high performance systems," *Cluster Computing: The Journal of Networks, Software Tools, and Applications*, vol. 4, no. 3, 2001.

[49] Engelen, R., "Code generation techniques for developing light-weight xml web services for embedded devices," in *ACM Symposium on Applied Computing*, 2004.

[50] Engler, D. R., Kaashoek, M. F., and O'Toole, J., "Exokernel: An operating system architecture for application-level resource management," in *Symposium on Operating Systems Principles*, pp. 251–266, 1995.

[51] Fabrice Bellard, "Qemu, a fast and portable dynamic translator," in *Proc. of USENIX ATC*, 2005.

[52] Fan, X., Ellis, C., and Lebeck, A., "The synergy between power-aware memory systems and processor voltage scaling," in *Proceedings of the Workshop on Power-Aware Computer Systems (PACS)*, December 2003.

[53] Foster, I. and Kesselman, C., *The Grid: Blueprint for a New Computing Infrastructure*. 1999.

[54] Foster, I., Kesselman, C., Nick, J., and Tuecke, S., ""grid services for distributed system integration"," *Computer*, vol. 35, no. 6, 2002.

[55] Fryman, C., Huneycutt, H., Lee, K., and Mackenzie, D., "Energy efficient network memory for ubiquitous devices," in *IEEE MICRO*, 2003.

[56] Ganev, I., *A Pliable Hybrid Architecture for Run-time Kernel Adaptation*. PhD thesis, College of Computing, Georgia Institute of Technology, 2007.

[57] Ganev, I., Eisenhauer, G., and Schwan, K., "Kernel Plugins: When a VM is too much," in *Proceedings of the 3^{r}d Virtual Machine Research and Technology Symposium*, May 2004.

[58] Gavrilovska, A., Kumar, S., Schwan, K., and Sundaragopalan, S., "Platform overlays: Enabling in network stream processing in largescale distributed application," in *NOSSDAV*, 2005.

[59] Gibbons, P., Karp, B., Ke, Y., Nath, S., and Seshan, S., "Irisnet: An architecture for a world-wide sensor web," *IEEE Pervasive Computing*, vol. 2(4), 2003.

[60] Gilman, L. and Schreiber, R., *Distributed Computing with IBM MQSeries*. 1996.

[61] Gribble, S., Welsh, M., Behren, R., Brewer, E., Culler, D., Borisov, N., Czerwinski, S., Gummadi, R., Hill, J., Joseph, A., Katz, R., Mao, Z., Ross, S., and Zhao, B., "The ninja architecture for robust internet-scale systems and services," *Computer Networks*, vol. 35(4), pp. 473–497, 2001.

[62] Grimm, R., Davis, J., Lemar, E., Macbeth, A., Swanson, S., Anderson, T., Bershad, B., Borriello, G., Gribble, S., and Wetherall, D., "System support for pervasive applications," *ACM Trans. on computer systems*, vol. 22, no. 4, 2004.

[63] Guerraoui, R. and Schiper, A., "Software-based replication for fault tolerance," *Computer*, vol. 30, no. 4, pp. 68–74, 1997.

[64] Gupta, N. and Das, S., "Energy-aware on-demand routing for mobile ad hoc networks," in *Workshop on Distributed Computing, Mobile and Wireless Computing*, 2002.

[65] Hagens, R., Hall, N., and Rose, M., "Use of the internet as a subnetwork for experimentation with the osi network layer," in *Request For Comments, RFC 1070*, 1989.

[66] Heinzelman, W., Murphy, A., Carvalho, H., and Perillo, M., "Middleware to support sensor network applications," *IEEE Network*, vol. 18(1), pp. 6–14, 2004.

[67] Hong, X., Liu, J., Smith, R., and Lee, Y.-Z., "Distributed naming system for mobile ad-hoc networks," in *ICWN*, 2005.

[68] Hossain, M. and Soh, W.-S., "A comprehensive study of bluetooth signal parameters for localization," in *IEEE PIMRC*, 2007.

[69] Howell, J., Jackson, C., Wang, H., and Fan, X., "Mashupos: Operating system abstractions for client mashups," in *USENIX HotOS*, 2007.

[70] Huang, Y. and Garcia-Molina, H., "Publish/subscribe in a mobile environment," in *ACM International Workshop on Data Engineering for Wireless and Mobile Access*, 2001.

[71] Hwang, J.-Y., Suh, S.-B., Heo, S.-K., Park, C.-J., Ryu, J.-M., Park, S.-Y., and Kim, C.-R., "Xen on arm: System virtualization using xen hypervisor for arm-based secure mobile phones," in *IEEE CCNC*, 2008.

[72] Isci, C. and Martonosi, M., "Phase characterization for power: Evaluating control-flow-based and event-counter-based techniques," in *Proceedings of the 12th International Symposium on High-Performance Computer Architecture(HPCA)*, February 2006.

[73] Jackson, C. and Wang, H., "Subspace: secure cross-domain communication for web mashups," in *Intl. conf. on WWW*, 2007.

[74] Jeff Dike, "User-mode linux," in *Proc. Linux showcase and conference*, 2001.

[75] Jejurikar, R. and Gupta, R., "Dynamic voltage scaling for system-wide energy minimization in real-time embedded systems," in *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)*, August 2004.

[76] Jennings, J., Whelan, G., and Evans, W., "Cooperative search and rescue with a team of mobile robots," in *International Conference on Advanced Robotics*, pp. 193–200, July 1997.

[77] Jim, T., Morrisett, G., Grossman, D., Hicks, M., Cheney, J., and Wang, Y., "Cyclone: A safe dialect of C," in *Proceedings of the USENIX 2002 Annual Technical Conference*, June 2002.

[78] `http://java.sun.com/products/jms`, accessed April 2008.

[79] Kang, P., Borcea, C., Xu, G., Saxena, A., Kremer, U., and Iftode, L., "Smart messages: A distributed computing platform for networks of embedded systems," *The Computer Journal, British Computer Society*, 2004.

[80] Kang, P., Borcea, C., Xu, G., Saxena, A., Kremer, U., and Iftode, L., "Smart messages: A distributed computing platform for networks of embedded systems," *The Computer Journal*, vol. 47, no. 4, 2004.

[81] `http://w3.antd.nist.gov/wctg/aodv-kernel`, accessed April 2008.

[82] Kong, J., Ganev, I., Schwan, K., and Widener, P., "Cameracast: Flexible access to remote video sensors," in *Multimedia Computing and Networking (MMCN'07)*, (San Jose, CA, USA), Jan. 2007.

[83] Kumar, R., Wolenetz, M., Agarwalla, B., Shin, J., Hutto, P., Paul, A., and Ramachandran, U., "Dfuse: A framework for distributed data fusion," in *ACM Conference on Embedded Networked Sensor Systems*, 2003.

[84] Kumar, S. and Schwan, K., "Netchannel: a vmm-level mechanism for continuous, transparentdevice access during vm migration," in *ACM VEE*, 2008.

[85] Kumar, V., Cai, Z., Cooper, B., Eisenhauer, G., Schwan, K., Mansour, M., Seshasayee, B., and Widener, P., "Implementing diverse messaging models with self-managing properties using iflow," in *International Conference on Autonomic Computing*, 2006.

[86] Lagar-Cavilla, H. A., Tolia, N., Satyanarayanan, M., and de Lara, E., "Vmm-independent graphics acceleration," in *VEE '07: Proceedings of the 3rd international conference on Virtual execution environments*, 2007.

[87] Lanfranchi, G., Peruta, P. D., Perrone, A., and Calvanese, D., "Toward a new landscape of systems management in an autonomic computing environment," vol. 42, pp. 119–128, 2003.

[88] LeVasseur, J., Uhlig, V., Stoess, J., and Gtz, S., "Unmodified device driver reuse and improved system dependability via virtual machines," in *Proc. of OSDI*, 2004.

[89] Li, Y., Wang, G., Yang, S., Shi, M., and Xu, J., "Research on grid-based cooperative platform," in *IEEE CSCWD*, 2002.

[90] Liu, H., Roeder, T., Walsh, K., Barr, R., and Sirer, E., "Design and implementation of a single system image operating system for ad hoc networks," in *International Conference on Mobile Systems*, 2005.

[91] Liu, J., Huang, W., Abali, B., and Panda, D. K., "High Performance VMM-Bypass I/O in Virtual Machines," in *Proc. of USENIX ATC*, 2006.

[92] Liu, T. and Martonosi, M., "Impala: A middleware system for managing autonomic, parallel sensor systems," in *Symposium on Principles and Practice of Parallel Programming*, 2003.

[93] Luckham, D. and Frasca, B., "Complex event processing in distributed systems," tech. rep., Stanford University Technical Report, 1998.

[94] Luo, J. and Jha, N., "Power-conscious joint scheduling of periodic task graphs and aperiodic tasks in distributed real-time embedded systems," in *International Conference on Computer Aided Design*, 2000.

[95] Luo, J. and Hubaux, J.-P., "A survey of inter-vehicle communication," Tech. Rep. IC/2004/24, EPFL, 2004.

[96] MacCormick, J., Murphy, N., Najork, M., Thekkath, C., and Zhou, L., "Boxwood: Abstractions as the foundation for storage infrastructure," in *OSDI*, 2004.

[97] Mascolo, C., Capra, L., and Emmerich, W., "Mobile computing middleware," *Advanced Lectures on Networking, LNCS*, 2002.

[98] Mascolo, C., Capra, L., Zachariadis, S., and Emmerich, W., "Xmiddle: A data-sharing middleware for mobile computing," *International Journal on Personal and Wireless Communications*, 2002.

[99] Mecella, M., Angelaccio, M., Krek, A., Catarci, T., Buttarazzi, B., and Dustdar, S., "Workpad: an adaptive peer-to-peer software infrastructure for supporting collaborative work of human operators in emergency/disaster scenarios," in *IEEE Collaborative Technologies and Systems*, May 2006.

[100] Menon, A., Cox, A. L., and Zwaenepoel, W., "Optimizing network virtualization in xen," in *Proc. of USENIX ATC*, 2006.

[101] Messer, A., Greeberg, I., Bernadat, P., and Milojicic, D., "Towards a distributed platform for resource-constrained devices," in *International Conference on Distributed Computing Systems*, 2002.

[102] Miyoshi, A., Lefurgy, C., Van Hensbergen, E., Rajamony, R., and Rajkumar, R., "Critical power slope: Understanding the runtime effects of frequency scaling," in *Proceedings of the 16th Annual ACM International Conference on Supercomputing*, June 2002.

[103] Nathuji, R., O'Hara, K., Schwan, K., and Balch, T., "Compatpm: Enabling energy efficient multimedia workloads for distributed mobile platforms," in *Proceedings of the ACM Multimedia Computing and Networking Conference (MMCN)*, 2007.

[104] Nathuji, R., Isci, C., and Gorbatov, E., "Exploiting platform heterogeneity for power efficient data centers," in *ICAC*, 2007.

[105] Necula, G. C., "Proof-carrying code," in *Proceedings of the* 24^{th} *Annual Symposium on Principles of Programming Languages*, pp. 106–119, ACM Press, 1997.

[106] http://www.netlab.cc.gatech.edu.

[107] O'Hara, K., Nathuji, R., Raj, H., Schwan, K., and Balch, T., "Autopower: Toward energy-aware software systems for distributed mobile robots," in *IEEE International Conference on Robotics and Automation*, 2006.

[108] Ooi, W., Pletcher, P., and Rowe, L., "Indiva: A middleware for managing distributed media environment," in *Multimedia Computing and Networking*, 2004.

[109] Ooi, W. T. and van Renesse, R., "Distributing media transformation over multiple media gateways," in *ACM Multimedia*, pp. 159–168, 2001.

[110] "Openbinder." `http://www.open-binder.org/`, accessed April 2008.

[111] Pai, A., Seshasayee, B., Raj, H., and Schwan, K., "Customizable multimedia devices in virtual environments," in *Intl. Workshop on Mobile Device and Urban Sensing*, 2008.

[112] Papazoglou, M. and Georgakopoulos, D., "Service-oriented computing," *Communications of the ACM*, vol. 46, no. 10, 2003.

[113] http://http://www.cc.gatech.edu/systems/projects/PBIO/.

[114] Pering, T., Agarwal, Y., Gupta, R., and Want, R., "Coolspots: reducing the power consumption of wireless mobile devices with multiple radio interfaces," in *MobiSys*, 2006.

[115] Perkins, C. and Royer, E., "Ad hoc on-demand distance vector routing," in *IEEE Workshop on Mobile Computing Systems and Applications*, 1999.

[116] Pike, R., Presotto, D., Dorward, S., Flandrena, B., Thompson, K., Trickey, H., and Winterbottom, P., "Plan 9 from Bell Labs," *Computing Systems*, vol. 8, no. 3, 1995.

[117] Pinheiro, E., Bianchini, R., Carrera, E., and Heath, T., "Dynamic cluster reconfiguration for power and performance," *Compilers and Operating systems for low power*, 2003.

[118] Pratt, I., Fraser, K., Hand, S., Limpach, C., Warfield, A., Magenheimer, D., Nakajima, J., and Mallick, A., "Xen 3.0 and the Art of Virtualization," in *Proc. of the Ottawa Linux Symposium*, 2005.

[119] Prehofer, C. and Bettstetter, C., "Self-organization in communication networks: principles and design paradigms," *IEEE Communications Magazine*, 2005.

[120] Raj, H. and Schwan, K., "O2s2: Enhanced object-based virtualized storage," in *SPEED*, 2008.

[121] Raj, H., Seshasayee, B., O'Hara, K., Nathuji, R., Schwan, K., and Balch, T., "Spirits: Using virtualization and pervasiveness to manage mobile robot software systems," in *IEEE International Workshop on Self-Managed Networks, Systems & Services*, 2006.

[122] Raj, H., Kumar, S., Seshasayee, B., Niranjan, R., Gavrilovska, A., and Schwan, K., "Enabling semantic communications for virtual machines via iconnect," in *International Workshop on Virtual Technologies in Distributed Computing*, 2007.

[123] Raj, H. and Schwan, K., "High Performance and Scalable I/O Virtualization via Self-Virtualized Devices," in *Proc. of HPDC*, 2007.

[124] Raj, H., Seshasayee, B., and Schwan, K., "Vmedia: Enhanced multimedia services in virtualized systems," in *Proceedings of Multimedia Computing and Networking*, 2008.

[125] Riva, O., Nadeem, T., Borcea, C., and Iftode, L., "Context-aware migratory services in ad hoc networks," *IEEE Transactions on Mobile Computing*, 2007.

[126] Rosu, D. and Schwan, K., "Faracost: An adaptation cost model aware of pending constraints," in *Real Time Systems Symposium*, 1999.

[127] Roth, C., "An introduction to enterprise javabeans technology," in *http://java.sun.com/developer/technicalArticles/ebeans/IntroEJB/index.html*, 1998.

[128] Schmidt, D. C., Gokhale, A., and Gill, C. D., "Patterns and performance of real-time and data parallel corba for high-performance embedded computing applications," in *HPEC*, 2002.

[129] Schmuck, F. and Haskin, R., "Gpfs: A shared-disk file system for large computing clusters," in *Proc. of FAST*, 2002.

[130] Schwan, K., Cooper, B., Eisenhauer, G., Gavrilovska, A., Wolf, M., Abbasi, H., Agarwala, S., Cai, Z., Kumar, V., Lofstead, J., Mansour, M., Seshasayee, B., and Widener, P., "Autoflow: Autonomic information flows for critical information systems," *Autonomic Computing: Concepts, Infrastructure, and Applications*, 2006.

[131] Seshasayee, B., Nathuji, R., and Schwan, K., "Energy-aware mobile service overlays: Cooperative dynamic power management in distributed mobile systems," tech. rep., GIT-CERCS-07-05, Georgia Tech, 2007.

[132] Seshasayee, B. and Schwan, K., "Mobile service overlays: Reconfigurable middleware for manets," in *ACM International Workshop on Decentralized Resource Sharing in Mobile Networks*, 2006.

[133] Seshasayee, B., Schwan, K., and Widener, P., "Soap-binq: High-performance soap with continuous quality management," in *icdcs*, 2004.

[134] Shah, R. and Rabaey, J., "Energy-aware routing for low energy ad hoc sensor networks," in *Wireless Communications and Networking Conference*, 2002.

[135] Shen, C., Ramamritham, K., and Stankovic, J., "Resource reclaiming in multiprocessor real-time systems," in *IEEE Transactions on Parallel and Distributed Systems*, 1993.

[136] Sivakumar, R., Sinha, P., and Bharghavan, V., "Cedar: a core-extraction distributed ad hoc routing algorithm," *IEEE Journal on Selected Areas in Communications*, vol. 17(8), pp. 1454–1465, 1999.

[137] Sivathanu, M., Prabhakaran, V., Popovici, F., Denehy, T. E., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H., "Semantically-Smart Disk Systems," in *Proceedings of the Second USENIX Symposium on File and Storage Technologies (FAST '03)*, 2003.

[138] Stoica, I., Morris, R., Karger, D., Kaashoek, M., and Balakrishnan, H., "Chord: A scalable peer-to-peer lookup service for internet applications," in *ACM conference on Applications, technologies, architectures, and protocols for computer communications*, 2001.

[139] Sugerman, J., Venkitachalam, G., and Lim, B.-H., "Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor," in *Proc. of USENIX ATC*, 2001.

[140] Swift, M., Annamalai, M., Bershad, B., and Levy, H., "Recovering Device Drivers," *ACM Transactions on Computer Systems*, November 2006.

[141] Swint, G., Pu, C., Koh, Y., Liu, L., Yan, W., Consel, C., Moriyama, K., and Walpole, J., "Infopipes: The isl/isg implementation evaluation," in *Network Computing and Applications*, 2004.

[142] Ta-Min, R., Litty, L., and Lie, D., "Splitting interfaces: making trust between applications and operating systems configurable," in *USENIX OSDI*, 2006.

[143] Takahashi, T., Sumimoto, S., Hori, A., Harada, H., and Ishikawa, Y., "Pm-2: High performance communication middleware for heterogeneous network environments," in *ACM/IEEE Supercomputing Conference*, 2000.

[144] Tennenhouse, D., Smith, J., Sincoskie, W., Wetherall, D. J., and Minden, G. J., "A survey of active network research," *IEEE Communications Magazine*, vol. 35, pp. 80–86, 1997.

[145] UPnP, "Universal Plug and Play forum." `http://www.upnp.org/`.

[146] Utz, H., Sablatnog, S., Enderle, S., and Kraetzschmar, G., "Miro - middleware for mobile robot applications," *IEEE Transactions on Robotics and Automation*, vol. 18(4), pp. 493–497, 2002.

[147] van Schaik, C. and Heiser, G., "High-performance microkernels and virtualisation on arm and segmented architectures," in *Intl. Workshop on Microkernels for Embedded Systems*, 2007.

[148] Vik, K.-H., Griwodz, C., and Halvorsen, P., "Dynamic group membership management for distributed interactive applications," in *LCN*, 2007.

[149] Vinoski, S., "Corba: integrating diverse applications within distributed heterogeneous environments," *IEEE Communications Magazine*, vol. 35, pp. 46–55, 1997.

[150] VMware, Inc., "VMware virtual platform, technical white paper," 1999.

[151] VMware white paper, "Understanding Full Virtualization, Paravirtualization and Hardware Assist." `http://www.vmware.com/files/pdf/VMware_paravirtualization.pdf`.

[152] von Eicken, T., Culler, D. E., Goldstein, S. C., and Schauser, K. E., "Active messages: a mechanism for integrated communication and computation," in *Proceedings of International Symposium on Computer Architecture*, 1992.

[153] Wallace, G. and others, "Tools and Applications for Large-Scale Display Walls," *IEEE Computer Graphics and Applications*, July 2005.

[154] Weiser, M., "The computer for the 21st century," *Scientific American*, vol. 265, no. 3, 1991.

[155] Woo, E., MacDonald, B., and Trepanier, F., "Distributed mobile robot application infrastructure," in *Intelligent Robots and Systems*, 2003.

[156] "Xenintro." `http://wiki.xensource.com/xenwiki/XenIntro`, accessed April 2008.

[157] Xu, D. and Jiang, X., "Towards an integrated multimedia service hosting overlay," in *MULTIMEDIA '04: Proceedings of the 12th annual ACM international conference on Multimedia*, 2004.

[158] Xu, M., Hu, Z., Long, W., and Liu, W., "Service virtualization: Infrastructure and applications," *The Grid: Blueprint for a New Computing Infrastructure*, 2004.

[159] Xu, R., Zhu, D., Rusu, C., Melhem, R., and Mosse, D., "Energy efficient policies for embedded clusters," in *Languages, Compilers and Tools for Embedded Systems*, 2005.

[160] Yang, C.-W., Lee, P. C. H., and Chang, R.-C., "Lyra: A system framework in supporting multimedia applications," in *Proceedings of IEEE International Conference on Multimedia Computing and Systems*, 1999.

[161] youb Kim, H. and Rixner, S., "Tcp offload through connection handoff," in *EuroSys*, 2006.

[162] Yu, C. and Chang, C., "Distributed query processing," *ACM Computing Surveys*, vol. 16(4), 1984.

[163] Zana, G., "Hvm pci passthrough." XenSummit, 2007.

[164] Zhang, X., McIntosh, S., Rohatgi, P., and Griffin, J. L., "Xensocket: A high-throughput interdomain transport for virtual machines," in *Middleware*, 2007.

[165] Zhang, Y. and Li, W., "An integrated environment for testing mobile ad-hoc networks," in *International Symposium on Mobile Ad Hoc Networking and Computing*, 2002.

[166] Zhong, L. and Jha, N. K., "Energy efficiency of handheld computer interfaces: limits, characterization and practice," in *MobiSys*, 2005.

[167] Zhou, D., Pande, S., and Schwan, K., "Method partitioning - runtime customization of pervasive programs without design-time application knowledge," in *ICDCS*, 2003.