Adaptive Protocol Management in Resource Constrained Devices

Mark Panahi, Sebastian Gutierrez-Nolasco and Nalini Venkatasubramanian
School of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3430, USA
Email: {mpanahi,seguti,nalini}@ics.uci.edu

Abstract—This paper focuses on techniques to achieve dynamic adaptation in the presence of device resource constraints. Specifically, we focus on achieving adaptive communication using dynamically constructed protocol stacks that allow applications to adapt to environmental and device changes with minimal resources. In the presence of resource constraints, we achieve this adaptation by dynamic switching of communication protocols to most effectively manage the limited device resources. Specifically, we propose 2 techniques, a simple demand-based strategy and a more sophisticated frontier based approach to determine which protocols to install/uninstall and when to trigger replacement. Intelligent protocol switching policies must also take into account protocol dependencies; we illustrate how this can be integrated efficiently into our algorithms. Our performance studies illustrate that the proposed strategies exhibit superior performance by reducing the overhead of protocol management while improving the messaging throughput of the system.

I. INTRODUCTION

Advances in mobile devices and wireless technologies are enabling pervasive computing environments, where small resource constrained devices execute a variety of applications. For example, consider the case of emergency personnel using multipurpose devices to monitor the environment, capture and process domain specific information. Applications executing on such devices have varying communication requirements (timeliness, reliability, security) implemented via communication protocols. Typically, the set of communication protocols that guarantees non-functional properties of applications on these devices may need to change based on changing levels of reliability, latency and security. A plug and play approach to protocol management (where protocols are uploaded when required) will ensure application requirements as device and network conditions change. Given resource constraints on hand-held devices, only a limited number of protocols can be active simultaneously. One solution would be to serialize the execution of applications on the device and load/unload the protocols required by the chosen application. Another solution is to switch applications when protocols required for its execution are not available. This batch processing approach results in poor application throughput and prohibits timely communication of information.

We believe that a viable alternative is to enable the concurrent execution of several applications by (un)installing communication protocols dynamically and efficiently based on individual communication requirements of applications in the system. In this approach, individual communication requirements are implemented as separate communication protocols, they are then combined to achieve specific application communication requirements. This alternative improves application performance and service response times due to fine-grain resource management since protocol replacement is transparent to the end-user application. Furthermore, we can expect that a given device using these mechanisms will perform as well as a more expensive device with greater resources that does not employ this approach. This fact implies that the application of this alternative will be especially useful in scenarios that require the massive deployment of such devices, due in large part to reduced unit costs.

In this paper, we present a framework for intelligent protocol management in resource constrained devices. We develop techniques to determine when protocols may be loaded or unloaded based on dynamic execution characteristics. Specifically, we propose 2 algorithms, a simple demand-based strategy and a more sophisticated frontier based approach to determine which protocols to install/uninstall and when to trigger replacement. The frontier based approach integrates past protocol behavior, current messaging requirements and future protocol demand to engineer an efficient policy for protocol replacement. We compare the proposed approach with a naive LRU-based policy and evaluate the conditions under which the adaptive protocol switching algorithms are useful. Such flexibility will enable the concurrent execution of applications with very specialized (and possibly contradictory) communication requirements in resource constrained devices without switching applications on-the-fly.

II. INTELLIGENT PROTOCOL REPLACEMENT

Consider a resource constrained device that services a set of concurrent applications. Each application establishes a list of communication protocols that needs to be applied to every incoming and outgoing message. Thus, we can visualize the system activity (from the communication subsystem point of view) as a set of messages \( M = \{m_1, \ldots, m_n\} \) and a set of protocols that might be used by any message \( P = \{P_1, \ldots, P_m\} \). Each message \( m_i \) has a priority \( m_i^\text{pri} \) and an ordered list \( m_i^\text{ord} \) specifying the execution order of the communication protocols that needs to be applied to the message. Every protocol \( P_k \) has a list of resource requirements,
Typically, we do not know in advance the time needed to process a message. Moreover, every message may be required to be processed with a different set of communication protocols. Due to the lack of resources, not all required protocols can be installed and executed concurrently. As a result, we observe that there are two independent subsets of protocols: those that are installed, denoted by \( P_{\text{install}} \subseteq P \), and those that are not, denoted by \( P_{\text{uninstall}} \subseteq P \). Protocols may be exchanged between the subsets at any time; however, there is an associated cost of conducting this exchange (if any) due in large part to the processing time required to capture and store the state of the protocol to be uninstalled, and possibly restore the protocol state if the protocol needs to be re-installed.

The specific ordering of protocols in a message’s protocol list creates a dependency between protocols for that message. In our work, we have developed intelligent heuristics for protocol replacement that attempts to capture protocol dependencies. Our goal is as follows: Given a set of messages \( m_1, \ldots, m_n \), each of which requesting to be processed with a particular list of communication protocols \( m_1^{\text{ord}}, \ldots, m_n^{\text{ord}} \), we would like to determine the set of suitable protocols to install such that current message priorities are obeyed and resource constraints are satisfied. Since protocol switches are expensive, our heuristics are designed to reduce protocol switches while ensuring efficient utilization of available resources.

III. POLICIES FOR PROTOCOL REPLACEMENT

An obvious choice to consider for a protocol replacement policy is LRU (least recently used) and other cache/page replacement policies. The LRU scheme and its variants LRU-MIN and LRU-THOLD [2] are very successful cache/page replacement policies due to the temporal locality property present in I/O request streams. However, protocol replacement is driven by message priority, protocol demand, and protocol dependencies, which vary according to the number of message arrivals and the list of protocols that every message specifies. Furthermore, in a protocol replacement policy, the state of a protocol may need to be saved immediately before it is uninstalled, and likewise, must be restored when the protocol is re-installed. Installation and uninstallation of protocols is further complicated by the fact that the size of the state held by a protocol can vary across time and across protocols. The temporal locality argument is difficult to make with protocol replacement because we cannot equate the number of reads/writes in a cache replacement to the number of messages in the protocol replacement scenario, since read/write of any word is a (small) constant number whereas the amount of time a message spends in a protocol is dependent on the application’s message and protocol.

We use a variant of the LRU policy as a baseline reference for our simulation studies. In our demand-based LRU variant, messages are ranked according to their priority and demand values and processed sequentially. Every protocol has a timestamp, which is updated every time it is executed; this allows us to identify the least recently used (installed) protocol. If an incoming message must be processed by an uninstalled protocol, then the least recently used protocol is uninstalled and the required protocol is installed in its place. In the case where the protocol to be installed requires more resources (e.g., memory) than the recently uninstalled protocol, our demand-based LRU variant will choose the next LRU protocol to be replaced. In other words, multiple protocols may need to be replaced to fit the needed protocol.

A suitable protocol replacement will attempt to install high demand protocols as early as possible; while keeping protocol management overhead small by reducing the number of protocol switches. In the following subsections we develop two policies to decide what protocols need to be (un)installed and when. The demand-based policy ranks protocols by priority and individual protocol demand values and uses this information to select the set of protocols with the highest priority and demand to be installed and executed. The Frontier-based policy takes into account the future (known expected demand) of protocols by potentially incorporating a look-ahead factor in the ordered protocol list for a message. In addition, it exploits knowledge of protocol dependencies in the ordered list of messages that are being processed. We further refine both policies by expanding the concept of protocol demand to include historical information.

A. Demand-based Policy

In this policy, we define the demand of a protocol as the number of messages ready to be executed by the protocol in the current cycle. Although some protocols are requested (or demanded) more frequently than others, their demand fluctuates since new message arrive (at varying message arrival rates) and existing messages are processed. Since there is a cost associated with every exchange, the idea behind this policy is to delay any protocol exchange (at the same or lower priority level) until the protocol demand of an uninstalled protocol significantly exceeds the demand of an installed protocol. We define two threshold exchange factors \( k_{\text{install}} \) and \( k_{\text{uninstall}} \) and define their values for each protocol \( P_k \) based on the percentage of messages processed by \( P_k \) in the past (e.g., historical information) and the associated cost of installing and uninstalling \( P_k \) respectively, both of which are obtained by profiling. \( k_{\text{install}} \) and \( k_{\text{uninstall}} \) allow us to influence an installed protocol to remain loaded and delay its installation if it is currently uninstalled. We constrain their values to be in a certain range defined by

\[
k_{\text{install}}^{P_k} = \sum_{i=1}^{n} \frac{\text{processed}(P_k)}{\text{processed}(\text{total})} + \text{install}(P_k)
\]

\[
k_{\text{uninstall}}^{P_k} = \sum_{i=1}^{n} \frac{\text{processed}(P_k)}{\text{processed}(\text{total})} + \text{uninstall}(P_k)
\]

Using this information, we create \( P_{\text{uninstall}} \) by selecting from \( P_{\text{uninstall}} \) the protocols with non-zero demand and order them first according to their priority value, and then by their demand
values. Then we rank the protocols in $P_{\text{install}}$ in the same way. For each protocol $P_{\text{needed}}$ in $P_u$, we iterate through $P_{\text{install}}$ until we find a protocol with the same or lower priority and less demand value. This protocol $P_{\text{candidate}}$ is a candidate to be uninstalled. We determine if the exchange needs to occur with the following relation: \[ \frac{k_{\text{available}}}{k_{\text{needed}}^\text{exchange}} < \frac{D(P_{\text{needed}})}{D(P_{\text{candidate}})} \] The exchange is carried out if and only if condition is true.

Since protocols may require different amounts of resources, it is possible that the protocol to be installed $P_{\text{needed}}$ requires more resources than the current uninstalled protocol $P_{\text{candidate}}$. In the particular case when this happens and there are no resources to carry on the switch, the next installed protocol with the lowest priority and demand $P_{\text{next}}$ is compared against $P_{\text{needed}}$. This process is repeated until a suitable replacement protocol can be found. If no replacement protocols are found, we cancel the installation process of $P_k$ (see Figure 1).

1) Create $P_u$ by selecting from $P_{\text{uninstalled}}$ the protocols with non-zero demand and order them according to their priority and demand values
2) Sort $P_{\text{install}}$ according to the protocols priority and demand values.
3) For each protocol $P_{\text{needed}} \in P_u$, iterate through $P_{\text{install}}$ until a protocol with the same or lower priority is found. Call this protocol $P_{\text{candidate}}$.
4) Determine if the exchange needs to take place by checking the replacement condition, \[ \frac{k_{\text{available}}}{k_{\text{needed}}^\text{exchange}} < \frac{D(P_{\text{needed}})}{D(P_{\text{candidate}})} \]
5) If the condition holds
   a) Save the state of $P_{\text{candidate}}$, if necessary
   b) Uninstall $P_{\text{candidate}}$
   c) While resources required by $P_{\text{needed}} >$ available resources
      - Iterate through $P_{\text{install}}$ until a protocol with the same or lower priority and demand values that $P_{\text{candidate}}$ is found. Call this protocol $P_{\text{next}}$ if it exists; otherwise cancel installation
      - Save the state of $P_{\text{next}}$, if necessary
      - Uninstall $P_{\text{next}}$
   d) Restore the state of $P_{\text{needed}}$, if and only if its state exists. That is, $P_{\text{needed}}$ was installed before and its uninstallation requires its state to be saved; otherwise just install $P_{\text{needed}}$

Fig. 1. Demand-based algorithm

The demand-based policy takes advantage of the current and historical protocol demand values and information respectively, in order to install protocols. However, it does not exploit protocol dependencies that create message demand flows between protocols.

B. Frontier-based Policy

In this section we introduce the concept of a frontier as the restricted set of possible protocols to be installed based on current protocol dependencies (in the ordered list of protocols for each message) and the set of installed protocols. Since every message already provides the ordered list of the protocols it requires to be applied to it needs, protocol dependencies can be exploited to pipeline the processing of messages at the same priority level. That is, given a set of messages, we iterate through their protocol list to determine the next group of protocols needed. The size of this group can be adjusted using a look-ahead factor $d$. With $d = 1$, protocols are installed according to their initial protocol demand and we use the message’s protocol list to influence the next immediately required protocol to be installed, pipelines the message from one protocol to the next on the list. A more aggressive approach groups protocols up to $d$ steps to identify the overall popularity of protocols. Using this popularity information a message processing plan is generated.

If $d$ is large, the frontier takes into consideration protocol required in the future (up to $d$ steps). If $d$ is small, only the immediate frontier is taken into consideration. Figure 2 illustrates the frontier. One motivating factor for using large values of $d$ is to exploit the existence of session-based communication. Messages associated with a session are typically processed using the same ordered protocol list. For example, in highly periodic sessions (e.g. streaming encrypted multimedia), a continuous stream of messages will require the same set of protocols. The frontier mechanism can exploit this session characteristics to group protocols into types and treat the set of protocols as a single type. Consequently, the demand of arriving messages is associated with protocol sets instead of individual protocols. Following the above argument, we classify messages in two categories: session-based and random. Session-based messages are required to be always processed by the same protocol ordered list ordl, producing a semi-continuous flow of message demand for those protocols; while random messages specify individual protocol ordered lists, which may potentially be different for each message. Thus, session-based messages protocol ordered lists are identified and protocol dependencies grouped in a straightforward manner in a master protocol ordered list ordl$\text{master}$ containing all the session-based ordered lists. Random messages are incorporated into this list on demand.

Fig. 2. Frontier Example

Figure 3 illustrates the frontier-based algorithm. The setup
phase initializes the algorithm; while the execution and update execute in an intertwined manner while there are messages to be processed in the system. Given a set of messages \( m_1, \ldots, m_n \), each of them requesting to be processed with a particular list of communication protocols \( m_1^{ord}, \ldots, m_n^{ord} \). The setup phase consolidates the individual protocol lists into a master protocol order list \( ordl_{master} \), ranked by priority level and initial demand values. Then, based on initial protocol demands, all protocols with non-zero demand are included as elements of the candidate set, denoted by \( candidate_{initial} \), representing the potential protocols to be installed. The initial set of protocols to be installed, denoted by \( PS_{initial} \), is selected based on priority and demand values from the candidate set and installed. Once the protocols have been installed, all protocols belonging to the candidate set and all protocol in the \( ordl_{master} \) that depend on any of them to process a message are identified (\( dependOn(PS_{initial}) \)) and included as elements of the frontier set, denoted by \( PS_{fronter} \), and are resorted according to their priorities and current demand values. When the update phase is triggered, the set of all possible protocols to install, denoted by \( candidate_{actual} \), is created by combining the elements of the current set of loaded protocols \( PS_{current} \) and their actual frontier \( PS_{fronter} \). The candidate list is sorted based on current priority and demand values. 

Since every protocol requires a certain amount of resources in order to execute, an alternate list, denoted by \( P_{2u} \), is constructed by parsing the candidates and keeping track of their resource requirements until the amount of resources surpass the available resources. That is, \( P_{2u} \subseteq candidate_{actual} \), such that resource constrains are maintained. Since the set of installed protocols and the set of candidate protocols to be installed may have common elements, the easiest way to identify the set of protocols to be uninstalled, denoted by \( P_{2u} \), is to subtract the set of candidates from the set of installed protocols. Once we know what protocols to uninstall, \( P_{2u} \), their state is saved and they are then uninstalled, and the set of protocols to install \( P_{u2i} \) are installed. The final step is to update the frontier (\( Frontier(P_{u2i}) \)) and the set of installed protocols.

IV. PERFORMANCE EVALUATION

In this section we describe the simulation environment and tested used to provide an initial performance evaluation of the policies discussed in the previous section under different workload conditions. Additionally, we provide some details of our prototype implementation.

Simulation Environment

Initial experiments were carried out in a simulated environment that we developed in Java. The simulation environment consists of one small resource-constrained device with 512K of available memory, running the Java runtime environment for resource-constrained mobile information devices (CLDC 1.0 and MIDP 1.0).

Due to the probabilistic and complex nature of traffic in communications systems, we use queuing theory to model message traffic and interarrival times, and network of queues to model protocol composition. In particular, we model every protocol queue as an \( M/M/1 \) queue. Thus, messages for a protocol arrive in a Poisson process at a rate \( \lambda \) and are served based on their priority.

In order to provide an accurate and fair performance measure of the different policies under different workload conditions, we use the message characterization defined in section III to provide three types of message inputs: (1) Session-based messages only, (2) Random messages only, and (3) Hybrid messages, which is a combination of 80% session-based messages and 20% random messages. Each message type and random message has it own \( \lambda \) values, which varies between 400 - 4000 ms.

In the simulation, we assume that there are 100 different protocols available in the system. Each one of them possibly consuming a different amount of resources ranging from 1 - 16KB, and not all of them are used in the simulation. Initially we assume that all protocols require the same amount of resources, but later we relax this assumption in order to explore the effects of protocol heterogeneity in terms of size.

Setup: Algorithm initialization

1) Create \( ordl_{master} \) by consolidating individual \( ordl \) lists
2) Create \( candidate_{initial} \) by selecting from \( ordl_{master} \) the protocols with non-zero demand and order them according to their priorities and current demand values respectively
3) Create \( PS_{initial} \) by selecting from \( candidate_{initial} \) a set of protocols to be installed, such that resource constrains are satisfied
4) \( PS_{fronter} = candidate_{initial} - PS_{initial} + dependsOn(PS_{initial}) \)
5) Sort \( PS_{fronter} \) according to their priorities and current demand values
6) \( PS_{current} = PS_{initial} \)
7) start protocol execution

Execution: Process messages until update is triggered

Update: 1) Update the set of possible candidates to be installed as follows:
   \( candidate_{actual} = PS_{current} \cup PS_{fronter} \)
2) Sort \( candidate_{actual} \) based on current priority and demand values
3) For each \( P \in candidate_{actual} \), add \( P \) as element of \( P_{2u} \), iff the overall requirements of \( P_{u2i} \) (including \( P \)) are less than the available resources
4) \( P_{2u} = PS_{current} - P_{u2i} \)
5) Uninstall protocols belonging to \( P_{2u} \)
6) Install protocols belonging to \( P_{u2i} \)
7) Update \( PS_{current} = P_{u2i} \)
8) Update \( PS_{fronter} = PS_{current} + frontier(P_{u2i}) \)

Fig. 3. Frontier-based algorithm
TABLE I
INPUT PARAMETERS USED IN THE SIMULATION

<table>
<thead>
<tr>
<th>Input parameter</th>
<th>Simulation values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum node capacity</td>
<td>40000 KB</td>
</tr>
<tr>
<td>Reconfiguration period</td>
<td>20 - 200 msec</td>
</tr>
<tr>
<td>Historical information</td>
<td>Yes / No</td>
</tr>
<tr>
<td>Total processed messages</td>
<td>2500</td>
</tr>
<tr>
<td>Lookahead factor</td>
<td>0 to 2</td>
</tr>
</tbody>
</table>

We also define 42 different types of messages that correspond to the available message types in the system. Each message type defines an ordered list of protocols and the size of the ordered list is variable (with a uniform distribution between 3 and 5).

We define input parameters (see Table I) and modify their values in order to observe their effect in the simulation. Node capacity is the amount of available resources on the node. We further define the term capacity ratio as the percentage of the node capacity that can be used to install and execute the communication protocols in the device. The reconfiguration period specifies how frequently the replacement policies are executed; while historical information allows us to specify if the policy will take into consideration recent past information in the exchange process.

Simulation Results

We measure the relative performance of the policies by running (for each policy) simulations with the same input parameters, but different message characterizations. We further analyze the results of these simulations under different message workloads (either varying λ values, the available resources in the node, enabling historical information, or all of them together). We have constructed three different input sets for the purposes of comparing these policies, as discussed above. All three experiments were repeated for various levels of resources.

We noticed that the behavior of the three policies vary markedly across input set types. Contrary to intuition, the LRU policy does not consistently underperform the other two policies at all times. For the session-based communication experiment (see Figure 4), the performance of the LRU policy roughly matches that of the other two policies at tighter resource levels. However, for higher capacities, the number of switches for both the demand and frontier-based policies reduce at a significantly faster rate than the LRU policy, with the frontier-based policy experiencing the most precipitous reduction. Once we introduce randomness to the input set (see Figure 5), we notice that the LRU policy behaves in a similar manner as in the previous experiment, and underperforms the other policies at all resource levels. We also observed that the performance of both the demand and frontier-based policies remain roughly constant across capacity ratios, with the frontier-based policy slightly outperforming the demand-based policy. For the purely random input set (see Figure 6), we once again observed that the LRU policy underperforms the other two policies, which in turn roughly match each other in terms of performance.

The Effect of Load

We explore the effect of load on the number of switches on the demand based policy by using random messages, set the capacity ratio at 50% and vary the interarrival rate of messages from 1 – 100ms. According to figure 7, we observe that there
is a lower bound at around 500 for switching as traffic intensity increases. Likewise, there is an upper bound at around 8000 for switching as traffic intensity decreases. Intuitively, at high traffic levels, messages tend to accumulate in the message queues associated with the various protocols. Once a protocol is installed, it will have the opportunity to process a large number of messages before subsequent uninstallation. In the low traffic case, this does not occur, and thus only a small number of messages can be processed within each protocol installation cycle.

Fig. 7. Switches as a function of load

The Effect of the Threshold Exchange Factor

As mentioned in section III, the threshold exchange factors, $k_{\text{install}}$ and $k_{\text{uninstall}}$, facilitate the control of protocol switching. In order to determine the impact of these values when the capacity ratio varies, we set $k_{\text{install}} = x$ and $k_{\text{uninstall}} = 1$ for all protocols, where $x = 1 \ldots 100$. Note that if $k_{\text{install}} = k_{\text{uninstall}} = 1$, it would be as if the parameter did not exist. As we can see from Figure 8, when the system is heavily loaded with messages (indicated by a 30% capacity ratio), there is the greatest improvement when $k$ is increased from 1 to 5, with little improvement thereafter. When the capacity of the system grows to better accommodate the installation of protocols (indicated by a 70% capacity ratio), we see that both values have virtually no effect (see Figure 9). The intuitive explanation behind this difference in results can be attributed to the fact that if message traffic is so low that the protocol demand in general stays close to zero, $k_{\text{install}}$ and $k_{\text{uninstall}}$ will have virtually no effect. In other words, when message traffic is low in relation to capacity, more resources will necessarily be available for switching, thus obviating the need for strict management of resources.

Period Selection

In our simulations we measure switches that occur (an installation/uninstallation pair), average and maximum message service time. Message service times are based on the time difference between the moment when a message enters the system from the application, to the moment when the message has been completely processed. Our primary metric for judging the performance of the policies is the number of switches produced by them during a given run of the simulation. For any given policy we can vary the frequency at which it executes (how often reconfiguration points occur). Variations in this frequency value produce different results in terms of the three primary performance metrics. For example, an increase in frequency may increase switching, but in turn reduce both service time metrics. Specifically, in Figure 10 and Figure 11 we observe that as the frequency decreases, service times rise sharply at first, but quickly level off. Likewise, the number of switches drops sharply but quickly levels off as well. Clearly, there is no clear choice as to the precise duration of the period, and is probably dependent on the resources available to the device.

Increasing the Look-ahead factor

The frontier-based policy that we have developed operates by constraining the protocol installation candidates to the currently loaded set of protocols and the frontier protocols. In general, the basis of this approach is not only to consider which protocols are currently required, but also what information we can glean with respect to future protocol requirements. When we refer to future protocol requirements, we are merely
considering those protocols that exist in the ordered protocol lists of all the messages in the system but are not part of the immediate requirements of a given message. In order to ascertain the influence of the look-ahead factor $d$, we vary the capacity ratio and the traffic load in one scenario using look-ahead factors $d = 0, d = 1$ and $d = 2$ and compare their performance in terms of switches (see Figure 12 and Figure 13) we see that in every case, increasing the look ahead factor from $d = 0$ to $d = 1$ decreases switching significantly, while a further increase to $d = 2$ provides a much less impressive marginal improvement. We conclude that the observed improvement is most likely due to the ability of this technique to capture the dependencies that exist in protocol lists associated with the messages currently in process.

**Protocol Heterogeneity**

We now relax the assumption that all protocols are homogeneous in terms of memory utilization and analyze the behavior of the system when exchanging protocols that vary in size. We assigned a state to each of the 100 protocols at initialization-time. The size of each state is random and uniformly distributed between $1 - 16KB$. Every time a protocol is installed or uninstalled, its state must be correspondingly restored or saved. Thus, we can surmise that the larger the state, the more costly the (un)installation process.

Note that we can no longer rely purely on the observation of the number of switches that occur of the course of the simulation. This is because the cost of an exchange may depend on the size of the state associated with a given protocol when it is both saved and restored. Therefore, we integrate the aggregate volume of data involved the exchange as a weight factor in the policy. The algorithm for considering protocol heterogeneity essentially operates to influence the mobility of a given protocol based on its corresponding weight. In other words, the demand for a protocol that is relatively heavy will be adjusted in such a way that it switches much less frequently than other protocols that are relatively light, assuming that both of them have similar demand values.

We use the frontier-based policy with a look-ahead factor of one, $d = 1$, and compare the performance differences with three different levels of available resources. From Table II, we observe that in general, factoring in weight information significantly reduces both switching, and the volume of data exchanged. However, we do notice that such a level of benefit is not as apparent when available resources are at a moderate level.
TABLE II

<table>
<thead>
<tr>
<th>Available resources</th>
<th>weight factor</th>
<th>switches</th>
<th>memory exchange</th>
</tr>
</thead>
<tbody>
<tr>
<td>25%</td>
<td>no</td>
<td>779</td>
<td>6327</td>
</tr>
<tr>
<td></td>
<td>yes</td>
<td>709</td>
<td>5618</td>
</tr>
<tr>
<td>50%</td>
<td>no</td>
<td>578</td>
<td>4724</td>
</tr>
<tr>
<td></td>
<td>yes</td>
<td>574</td>
<td>4692</td>
</tr>
<tr>
<td>75%</td>
<td>no</td>
<td>443</td>
<td>3563</td>
</tr>
<tr>
<td></td>
<td>yes</td>
<td>405</td>
<td>3216</td>
</tr>
</tbody>
</table>

V. PROTOTYPE IMPLEMENTATION

To illustrate the feasibility of our ideas, we use the ComPOSE—Q middleware framework [3] as our run-time environment and a reflective communication framework (RCF) [4] as our communication environment to test our policies under real-time conditions. The prototype implementation provides a first step towards a cost-effective communication framework capable of addressing adaptive environments with evolving policies.

In the RCF each object has a messenguer, which customizes its communication and there is one communication manager in every node of the distributed system, which implements and controls the composition of communication protocols specified by messengers on that node. The communication manager has a set of communication protocol objects, each of them implementing a particular communication protocol provided by the framework (e.g., reliable, in-order or security). This scheme allows us to abstract a core set of communication protocols and share it between the different messengers on a node, simplifying the synchronization and composition process. Furthermore, it encourages separation of concerns in the process of message transmission and reception. In order to provide an efficient implementation, the communication manager implements a set of message coordinators. Every message is assigned to a message coordinator, which handles the protocol execution order specified in its protocol list. This concept of reusable message coordinators is an efficient way to handle the service request of each messenger without having to pay the bottleneck associated with the centralization of the services in the node communication manager and allows us to concurrently process multiple messages. In the RCF, a communication protocol may be explicitly requested (at any instance) by the sender or implicitly specified by its messenguer, in case the sender and the receiver previously agreed to use a particular communication protocol. In order to provide dynamic customization of communication protocols (potentially on a per message basis), the RCF model separates specification, composition and implementation of communication protocols. The specification is handled by the messenger through a message service list (msl), which determines the set of communication protocols to be used in the communication of a specific message to its target. The composition is handled by the message coordinator via the creation of a composition order list (ordl), which may constrain protocol execution in order to obtain the desired combined benefits. How this ordl is created has been discussed in [4].

A. Protocol Management Module

Since protocols are implemented by objects, protocols can be installed or uninstalled dynamically using the protocol management module (PMM), which is composed of a message queue cache, a protocol holder and a protocol loading manager (see Figure 14).

The message queue cache, mqc, is responsible for storing message queues for the different communication protocols (installed or not). Since the system should give preference to those protocols that are the most urgently needed and secondarily to those that are the most burdened with messages, we define the following two terms, which are stored along with the message queue: (i) Criticality - the priority of the highest-priority message queued for each protocol, and (ii) Demand - the number of highest-priority messages queued for a protocol.

When a protocol is uninstalled, its state is captured and stored in a remote or local repository. The protocol holder ph holds a reference to its location in case the protocol needs to be reinstalled. The Protocol Loading Manager plm enacts the current policy in order to determine what and when a protocol should be installed or uninstalled.

![Fig. 14. The reflective communication framework and the protocol management module](image)

B. Prototype Evaluation

The platform used to measure the performance of the prototype implementation were Sun Ultra5 workstations (333Mhz UltraSPARC IIIi with 256KB external cache and 128 MB RAM) running Solaris 2.7, IBM thinkpad laptop (700 Mhz Pentium III with 256MB RAM) running Windows XP and a Compaq iPaq 3650 (206 Mhz StrongArm with 16MB ROM, 32MB SDRAM) running Linux (PocketLinux distribution) in the 16MB ROM. The workstations were connected via a 100Mb/s Ethernet link; while the laptop used a wireless Cisco 350 client card and the handheld device a Lucent Orinoco Silver 11Mbps wireless PCMCIA network interface. The performance metric results presented below were obtained without timing in-lined JVM internal calls using J2SDK1.4.1, J2ME Wireless toolkit 2.0, CLDC 1.0 and MIDP 2.0.

The main goal of this preliminary results were to obtain the basic communication overhead of the RCF in mobile constrained devices. In order to measure the end-to-end message overhead, we categorized the overheads involved in the process (see Table III). First we measured the message transmission...
and reception overheads of the underlying transport layer. Then, we measured the time needed to send messages through the RCF. Finally, we send processed messages using a reliable secure protocols and measured the time needed to deliver it to the target object.

VI. RELATED WORK AND CONCLUDING REMARKS

Current middleware infrastructures have incorporated the notion of reflection in order to provide the desired level of configurability and openness in a controlled manner [5] [6]. However, they do not deal with dynamic installation and uninstallation of communication protocols, e.g. The pluggable protocol framework [7] addresses the lack of support for multiple inter-ORB protocols and deals with integration and use of multiple ORB messaging and transport protocols; while DynamicTAO [8] explore ways to make the various components of an ORB dynamically configurable as well as componentizing them to achieve minimal footprint for small applications [9]. In recent years, several group communication systems have been developed [10] [11] [12] [13] [14] to support modular and reconfigurable group communication, but the protocol stack can only be configured from a selected protocols and dynamic installation and uninstallation of communication protocols are cumbersome and error-prone.

In this paper we propose 2 techniques, a simple demand-based strategy and a more sophisticated frontier based approach to determine which protocols to install/uninstall and when to trigger replacement. Future work includes a more careful understanding of the impact of protocol resource consumption in the exchange process.

REFERENCES