Practical Experience with Adaptive Service Access

Umar Saif
umar@mit.edu

Justin Mazzola Paluska
jmp@mit.edu

Vijay Praful Chauhan
chauhan@stanford.edu

MIT Computer Science and Artificial Intelligence Laboratory
Cambridge, MA 02139 U.S.A.

The dynamically changing nature of the emerging networked environments warrants a computing model in which resources are dynamically discovered and opportunistically utilized to maintain continuity of service. This paper describes the design and implementation of Lightweight Adaptive Network Sockets (LANS) for accessing services in such a dynamically changing networked environment. A LANS socket takes a high-level description of a service and opportunistically connects to the best provider of the service in the changing environment of the application. LANS builds on the architectural philosophy of Service-oriented Network Sockets (SoNS)[13], in that it integrates a service-oriented abstraction with the operating system socket interface and provides adaptive service access at the end-host session layer. However, our experience with SoNS led to three key improvements for the LANS architecture. 1) The LANS session layer reduces the computation and communication overhead resulting from the SoNS end-to-end architecture. 2) LANS is designed to offer richer semantics for resource selection and allocation to enable better utiltization of resources in shared pervasive environments. 3) Applications in LANS can control the opportunistic behavior of the system to avoid perfunctory reconnections. Our experiments show that LANS running on an iPAQ consumes 40% less power, requires less than 1% of the network traffic, and is far less processor hungry than the original version of SoNS — without compromising performance and offering richer semantics for adaptive service access.

I. Introduction

Recent years have seen a growing interest in pervasive computing environments [4, 5]. Peppered with wireless, mobile and embedded devices, such environments are characterized by a degree of dynamism not common in traditional distributed systems. Applications in such a dynamic environment must cope with mobility of devices, rapid fluctuations in wireless interconnects and frequent, typically abrupt, changes in available resources. A fundamental problem highlighted by such emerging environments is how to satisfy and sustain an application's service requirements in the face of continuous changes in available resources. In such an environment, resources available to satisfy a service request depend heavily on the details of the runtime environment, and may be impossible to anticipate prior to the service request. Furthermore, the extent of an application-level function may go beyond the availability of any specific resource, while the suitability of a resource to satisfy the application service requirements could change during the lifetime of the application. An application in such a system, therefore, cannot be programmed to depend on a predetermined set of resources to access a service. Rather, resources must be dynamically discovered and opportunistically utilized to sustain the application's service requirements in its changing environment.

Typically, an application in such a diverse and dynamic environment can only provide an approximate description of the service it expects from its environment, while the underlying system must provide access to the most appropriate service provider. Such a system must be able to continuously adapt to changes in user locations and needs, respond both to component failures and newly available resources, and maintain continuity of service as the set of available

resources changes. The task of accessing the best provider of a service often entails a certain degree of planning involving continuous (re-)evaluation of available alternatives, as well as heuristic compromises to best address the application's requirement using imperfect resources in its changing environment.

In the past, the functionality required for adaptive service access had to be implemented as part of the application. Hence every application must be burdened with the additional chore of continuously discovering the available resources, evaluating and comparing their suitability, and opportunistically accessing the resource that best matches its requirements. With the increasingly dynamic nature of emerging networked environments, it is clearly desirable to offer such adaptive service access as part of the underlying communication primitives for accessing networked services.

The rest of this paper is organized as follows. In section II we review two adaptive service access systems, Intentional Naming System (INS)[7] and SoNS. Next, in Section III, we outline the lessons we learned from using these systems as part of Project Oxygen [12]. Sections IV and IV.B describe an extended version of SoNS that addresses the shortcomings of previous systems. Section V evaluates our new architecture. In section VI we summarize related work, and finally in section VII we conclude the paper.

II. Approaches to Adaptive Service Access

A system for providing adaptive service must encompass the following three principal mechanisms:

Service-oriented Communication An application must be able express its requirements in terms of the service it expects rather than a specific server it must access to use the service.

Resource Discovery and Selection Given a service description, the system must be able to dynamically discover the available resources, evaluate their suitability against application requirements, and connect the application to the resource that best satisfies the service requirements.

Opportunistic Access To maintain a continuity of service in a dynamically changing networked environments, the system must be able to opportunistically access the best available service provider. For this, the system must continuously monitor the environment for better alternatives, and reconnect the application if a better alternative becomes available.

These mechanisms can be embedded at different layers of system design. In this section we report on our experience with two different architectural approaches to providing system-level adaptive service access: Intentional Naming System (INS)[7] and Service-oriented Network Sockets (SoNS)[13].

The Intentional Naming System integrates all three mechanisms required for adaptive service access with datagram routing at the network layer. It does this using an overlay network of Intentional Name Resolvers (INRs). A service joins the network by communicating its description with an INR. In turn, the INR informs the rest of the overlay network of the new service. Applications communicate with network services using *intentional datagrams*. Each *intentional datagram* is marked with a service description, while the the overlay of INRs route the datagrams to the service-provider most closely matching the datagram's service description. Adaptive service access follows naturally: as service-providers come-up or disappear, the INS overlay simply forwards the datagrams to the best matching service-provider at any given time.

The key strength of a network-layer approach like INS is its simplicity: an applications simply tags its packets with the description of the desired service, while the routing layer hides the complexity of delivering it to the best available service-provider.

However, this simplicity is also a major weakness of the INS approach. Since the logic for selecting the best match for the application is hidden in the routing infrastructure, an application has little control over what gets accessed. Worse, the INS approach of integrating resource discovery with datagram routing inherently lacks session-semantics; since each datagram is routed independently, two successive datagrams can be routed—transparently to the application—to two different service-providers.. This lack of session-semantics precludes a large body of mobile applications, including typical *context-aware* applications like "follow-me-video", and, can lead to thrashing between service-providers in the presence of characteristic fluctuations in a wireless network. From a performance point of view, the approach of resolving service-descriptions in the critical path of message delivery is orders of magnitude slower than traditional IP-based routing (10^3 datagrams/second in INS vs 10^8 packets/second for typical IP routing). Moreover, tagging each packet with a textual representation of the service-description wastes precious bandwidth in a wireless network. Finally, even with a robust fault detection and recovery mechanism, it is often cum-

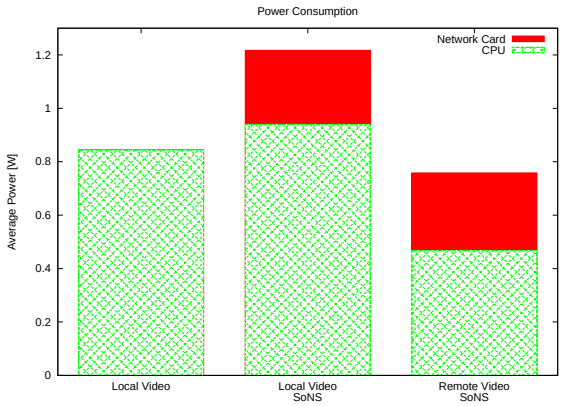


Figure 1: Power Consumption in Watts for "follow-me-video" application. The overhead of SoNS results in approximately 50% more power to be consumed when the video is playing locally, while resulting in a net loss of power even when the video is offloaded to an external display (and the iPAQ screen is shutoff)

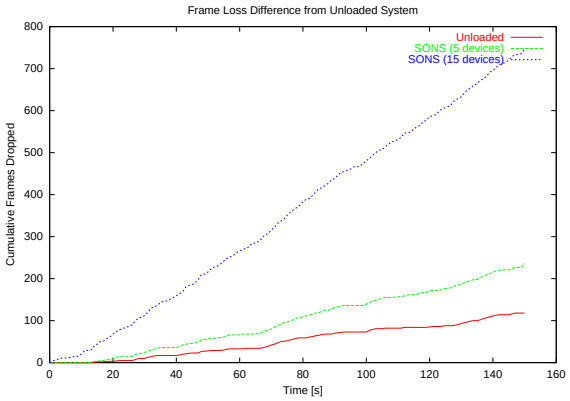


Figure 2: The number of frames dropped by a media player (playing an MPEG of the Matrix 2 trailer) on an iPAQ using SoNS. The crosstalk of SoNS with the media player becomes more noticeable as the number of matching display devices is increased from 5 to 15.

bersome to setup and maintain an overlay of resolvers required for INS' correct operation.

In our previous work, we propose Service-oriented Network Sockets (SoNS) [13] for accessing services in a dynamically changing networked environment. Instead of employing a content-based routing overlay like INS, SoNS is an end-host system, interposed at the session-binding layer. SoNS extends the traditional operating system socket interface to offer session-oriented semantics. An AF_SONS socket is *connect()*ed by an application by providing an abstract description of the desired *service* (rather than the network address of a specific *server*). Subsequently, the SoNS session layer opportunistically accesses the best available service-provider by continuously discovering the available resources, evaluating their suitability against application's service requirements and (re-)reconnecting the application session to the resource that best satisfies the service description. We refer to this periodic discover-evaluate-rebind cycle as the *adaptive session loop*.

Unlike content-based routing approaches, SoNS defines an end-to-end architecture [14], and consequently offers applications more control over the session rebinding semantics. SoNS permits an application to configure the *context* within which the application is interested in finding the service-providers, the *agility* with which the system should probe the environment for better alternatives, and a measure of *hysteresis* that specifies for how long a better alternative must be available before the system considers rebinding. Finally, an application can register a *callback* that is invoked by the SoNS session-layer to notify the application about a potential rebinding. The application can use the callback to prepare for rebinding its network session or to even reject the rebinding all together. It is noteworthy, however, that all of these parameters are optional, specified using the *setsockopt* interface. Therefore, simpler applications may configure only a subset of these parameters, or even leave them unspecified, causing the system to use default values.

Since SoNS addresses the dynamism of a mobile environment at the end-host session-binding layer, it does not require a special routing framework, does not introduce the cost of resolving service descriptions in the critical path of the application and does not require every network message to carry the description of the required service. Importantly, unlike a routing-based system like INS in which fault-recovery happens in the critical path of service access—even if one of the routing nodes malfunctions, the application cannot access the service-provider until the INS overlay re-computes its routing spanning tree—the end-to-end architecture of SoNS does not require any additional mechanism to setup and maintain other hosts in the system for accessing a service-provider.

III. Lessons Learned

Though the session-oriented end-to-end architecture of SoNS has several advantages over content-based routing approaches, our experience with adaptive applications led us to reconsider a number of design

choices in SoNS. In this section, we present the key insights that emerged from our experience of using SoNS as part of Project Oxygen[12].

End-to-end session-semantics may cause a large overhead in a mobile environment. Our experience revealed that SoNS's combination of an end-to-end design and an application-customizable session-layer architecture leads to an unacceptably high overhead both at the client and the service-providers embedded in a pervasive computing environment.

To illustrate by example, consider the overhead of SoNS for a popular context-aware application in our environment: "follow-me-video". This application, intended for devices like portable DVD players, redirects the video-stream playing at a mobile device to a large nearby display for improving the viewing quality of the video and to save resources on a mobile device. However, while SoNS makes it extremely simple to write such an application, we found that running an adaptive session loop on a low-end mobile host not only results in significantly more consumption of battery power (Figure 1), but also may start interfering with the application itself (Figure 2).

Furthermore, while SoNS' adaptive session loop can be customized to suit the application-level session semantics, treating every application session independently at each client often results in redundant discovery messages. For instance, five applications wishing to access the *colored printer on the 6th floor*, will each generate five different probes every *AGILITY* seconds for the same printer, resulting in increased overhead for service-providers.

"Best-match" is not always the best match in a shared environment In SoNS, an application expresses its service requirements as a set of constraints on the properties required in an ideal resource, while the system attempts to satisfy the request with the resource that best satisfies those constraints. For instance, an application wishing to access a display with a size larger than 15 inches (size ≥ 15) is connected to the largest available display which is at least 15 inches in size. Similarly, an application wishing to access a printer with a print-queue of less than 4 is given access to the least loaded printer which has at most 4 items in its print-queue. Once an application has access to a resource, it cannot be reclaimed to satisfy another application until the first application disconnects from it (assuming the resource cannot be multiplexed between applications).

While these simple semantics were adequate for our early prototype applications, the use of SoNS in a growing body of applications in shared spaces quickly forced us to reconsider the resource selection and allocation policy. To illustrate the problem with the SoNS approach, as well as with INS and other related systems, consider a simple scenario in which two applications want to access a display in an environment equipped two displays of size 17 and 28 inch respectively. The first application invokes SoNS with a service request of "size ≤ 15" and, assuming that both the displays are available, the application is provided access to the 28 inch display. Shortly after, the second application requests for a display of size larger than 25 inches. Given the resource selection and allocation policy of SoNS, the second request will fail even though the available resources could satisfy both the applications. Clearly, as more devices use opportunistic service access, such a greedy approach to service allocation leads to an increasingly poor utilization of shared resources.

Opportunistic access may cause perfunctory reconnections SoNS is designed to automate the task of opportunistically accessing resources that best match application requirements. A downside of this automation is that such a system may cause new a service-provider to be chosen even when the marginal benefit of the new selection is imperceptible to the application; SoNS simply triggers a rebinding when the score assigned to a new resource is better than that of the current resource for a consecutive HYSTERESIS number of probes. As a result, we frequently found ourselves putting extra code in the application-callback to calculate the improvement afforded by every new selection. This not only introduces additional complexity in the application, but also leads to many suggested resources being rejected, resulting in wasted computation.

IV. Extending SoNS

In this paper we present the design and implementation of an extended version of SoNS. This version, called Lightweight Adaptive Network Sockets (LANS), improves SoNS in four key dimensions:

1. LANS reduces the overhead at a mobile client by using a replicated server framework while still preserving the session-oriented semantics and robustness of SoNS.
2. LANS extends the simple "best-match" semantics of resource selection and allocation to im-

prove the utilization of shared resources in a pervasive environment. In addition to a simple "best" match, LANS offers two flavors of "less-than-best" matching: "barely-sufficient" matching for a conservative allocation of shared resources between competing applications, and "best-but-loaned" mode in which the system can dynamically reallocate the available resources to satisfy cooperating applications in a shared space.

3. LANS provides an application greater control over the opportunistic behavior of the system; unlike SoNS, LANS can be configured by an application to rebind only when a new service-provider exceeds the application's perceived cost of rebinding.
4. LANS collates similar application requests to amortize the overhead of periodic discovery, while still permitting individual applications to tailor the semantics of session rebinding.

IV.A. LANS Server Framework

To reduce the overhead of SoNS, without comprimising application-level end-to-end semantics, we choose a very practical and simple approach in LANS: instead of having every client independently execute a periodic discover-evaluate-rebind cycle, LANS offloads the adaptive session loop to shared *LANS servers* in the environment of the client. However, even with this simple, and seemingly conventional approach, the LANS system architecture must address several interesting challenges to preserve the robust, end-to-end semantics of SoNS.

Robustness A key strength of SoNS is that each client is embedded with complete functionality required to provide adaptive service access. To maintain the same level of robustness, LANS must be robust against the failure of LANS servers running the application request.

Scalability In order to support a large number of potentially diverse application requests, each configured with its own session-semantics, a shared server framework must be scalable and permit localized discovery and monitoring of resources.

Relativity Though a client can offload the discovery and evaluation of service-specific attributes to a server, some attributes are only meaningful when computed end-to-end, such as network-sensitive or location-based attributes.

Like SoNS, LANS is interposed at the end-host session layer. However, in order to minimize the load on a mobile client, the LANS session layer can offload the overhead of running the adaptive session loop to a LANS server. The LANS server, in turn, calls the client with a rebinding suggestion only when it discovers a resource matching the application requirements.

At the same time, even with this server-based optimization, LANS preserves the robustness of SoNS. Each LANS client is supported by a replicated set of servers, while the request-handover protocol between the client and the server framework is designed to be fail-safe. Fail-safety in LANS is provided by a two-tier failover protocol: failure of a LANS server causes the application request to automatically failover to another suitable server, while, more importantly, in an environment in which no suitable server can be used to handover the application request, the system fails-over to the client itself, operating similarly to SoNS to maintain a continuity of service. In other words, LANS is optimized to save resources on a mobile client by taking advantage of a replicated server framework, but the correct operation of the system is not dependent on the availability of external servers.

In order to define a scalable, localized discovery model, each LANS Server is associated with a discovery *context*, which is matched against the context specified in the application request when selecting an appropriate server for offloading the request. A server's context may be specified either manually or auto-configured by a geographic location system.

Finally, the LANS evaluation framework makes a distinction between resource-specific attributes, e.g. [service-type= Display, Size$\geq$15 inches], and network-sensitive attributes, e.g. [latency $<$ 200 ms]. Resource-specific attributes are passed to a LANS server, while attributes that are only meaningful when computed end-to-end are computed at the client.

Enhanced Application Control LANS defines two additional socket options to permit greater application control over the selection and rebinding to resources. The first option, SO_NICE, configures the system for less-than-best matching in shared spaces, while the second option, SO_IBT (if-better-than), permits the application to specify the degree of improvement it perceives worthwhile for migrating a session to a new resource.

Less-than-best Resource Selection: LANS supports two less-than-best resource selection mechanisms depending on whether the system is operating in failover or server-assist mode. In server-assist mode, the server acts as the shared

arbitrator between competing applications. The SO_NICE option puts LANS in "best-but-loaned" mode, which is satisfied by connecting the application to the best available resource but with the possibility of a subsequent downgrade to accommodate a new request. In this mode, therefore, an SO_NICE application can potentially end up using the best available resource if no other application makes a conflicting request. In failover mode, however, since there is no shared arbitrator for resource-allocation, the system to must operate in a more conservative fashion. A SO_NICE request in failover mode, called "barely-sufficient-matching", is satisfied by simply connecting the application to the resource which barely satisfies its requirements, leaving better alternatives to be potentially used by more stringent applications later on. It is worth noting that since the SO_NICE option is satisfied by the least-acceptable resource, it in fact does not require subsequent probing for better alternatives. Neither is this conservative mode satisfied by opportunistically accessing poorer matches. Rather, the application is simply connected to the least-acceptable resource at the time of the request, while a rebinding is triggered only if the originally selected resource becomes unavailable. The SO_NICE option, therefore, also makes the other socket options irrelevant in the failover mode i.e. SO_CONTEXT, SO_HYSTERESIS, SO_AGILITY.

Utility-based Resource Rebinding: In order to avoid perfunctory reconnections, an application using LANS uses the SO_IBT option to specify the factor of improvement that must result from a connection-rebinding before the system invokes the application-callback with a rebinding suggestion. The factor of improvement, k, is specified as a floating-point number, e.g. 2.5, while the rebinding is suggested if and only if the following test evaluates to true:

$$score_{new} \geq \sum_{n=i}^{i+h} \frac{score_n}{h} * k$$

where h is the HYSTERESIS value for the socket.

When configured with the SO_IBT option, the LANS parser/worker module suggests a rebinding only when a new resource achieves the highest score for the HYSTERESIS number of probe cycles and its score is an improvement over the current selection by at least the factor specified in the SO_IBT option.

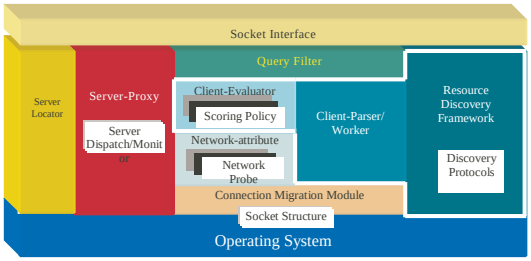


Figure 3: LANS client architecture.

IV.B. LANS System Architecture

Figure 3 shows the architecture of a LANS client. Like SoNS, LANS exports the OS socket interface for configuring and establishing a service-oriented network session. However, unlike SoNS, a LANS client has the option of handing-over the service specification to a nearby LANS server instead of handling it locally.

The LANS client architecture comprises of two blocks of functionality. The block demarcated by a white border in the figure (comprising of Client-Parser/Worker, Client-Evaluator and a Resource Discovery Framework) encapsulates the functionality that is only invoked when a LANS server is not available, causing the system to failover to the client itself and operate similarly to SoNS. This block is henceforth referred to as the *failover block*. All other components in the system (including the Server Locator, Server Proxy, Network-attribute Evaluator and the Connection-migration module) are used in the routine operation of the system. When a LANS server is available, the Server-proxy module is responsible for offloading the process of discovering, monitoring and comparing the available service-providers to the server.

Below we detail the LANS client-server interaction. Since the LANS failover block is same as SoNS, albeit with support for "barely-sufficient" and "utility-based" resource selection mentioned above, it is not described further in the paper.

IV.B.1. Server-assisted Mode

When the Server-locator module finds a LANS server, the socket-wrapper passes the request to the Server-proxy with the address of the LANS server returned by the server-locator module. Once invoked, the

server-proxy establishes a long-lived TCP connection with the specified LANS server and sends a SERVER_REQUEST message to the client, containing the resource-specific constraints from the service description, application-supplied session-rebinding parameters, and the maximum number of initial matches, N, to be returned by the server. The reason for requesting a list of top N matches, dubbed the n-best-list, instead of simply the most suitable resource, is to provide sufficient choice to the client-side network-attribute evaluator for making a selection based on the application's network constraints; N is simply set to 1 when the application request does not include any network-sensitive attributes.

In the background, the LANS server continuously monitors the client environment, and notifies the Server-proxy if a better alternative becomes available. On receiving such a notification by the Server, the Server-proxy triggers the cycle to evaluate the network-attributes of the suggested resource. Finally, if the newly suggested resource satisfies the network-sensitive constraints, the Server-proxy requests the connection-migration module to reconnect the underlying socket to the suggested resource (after seeking permission from the application by invoking the application-callback), and informs the Server that its suggestion was accepted. The Server-proxy could also deny a rebinding suggestion if the network-attributes of the suggested resource do not satisfy the application-constraints or the application prefers not to have its session rebound to another resource.

LANS network-attribute evaluator is based on an extensible design, permitting "network-probes" to be added at runtime e.g. add_probe("latency", /usr/sbin/ping). Subsequent application requests are then matched against the list of registered network-probes, while network-attributes are removed from the request and computed at the client – without requiring any special notation on the application's part to identify the network-attributes in its request.

The LANS server-locator module keeps track of the available servers, triggering a switch over to the server-assisted or the fail-over mode depending on the availability of a LANS server. In order to minimize the overhead of LANS server discovery, and to speed-up server failover (described later) the server-locator module maintains a cache of discovered servers, indexed by their context.

IV.B.2. Fail-safe Request-handover

The LANS server proxy is responsible for offloading an application request to a LANS server. To accomplish this, the LANS server-proxy forwards the resource-specific constraints to the LANS server returned by the server-locator, monitors the health of the server, and provides automatic failover between servers if the currently employed server becomes unavailable.

The key architectural consideration in the LANS failover architecture is the allocation of responsibility between the client and the servers. In a system where the onus of failover is placed on the servers, like the IETF Reliable Server Pooling architecture [15], a server failure causes another server to automatically and transparently take over the responsibilities of the failed server. However, in this scheme, every server must typically be instrumented with the protocols and mechanism for detecting a failed server, electing an appropriate replacement for the failed server, and transparently migrating the client session to the new server.

However, keeping with the end-to-end architectural semantics of our approach, server-failover in LANS is provided by the client. This not only provisions the semantics for migrating application-state where the application itself resides – at the client – it avoids the complexity associated with server coordination and election by using a simple mechanism in the client's server-proxy module. The client-side server-proxy provides failover by simply monitoring the health of the currently employed server and migrating the application service-request to another suitable server if the current server fails to respond to a health probe. Since the discovery of an alternative server typically comes for free from the server-locator cache, the overhead of providing failover in this case is simply the cost of periodic health probes from the client to the server. Importantly, since this end-to-end architecture exposes a server fault to the client, rather than hiding it in the intermediary server pool, it naturally enables client-failover to handle the case when an alternative server is not available to take-over the application service request.

The server-proxy leverages TCP keepalive (SO_KEEPALIVE timer) with the LANS server to enable periodic health probing. The overhead of periodic probing is fine-tuned to suit the application requirements by carefully choosing the probing interval of the keepalive. Specifically, the probing interval is set to a value slightly less than the *AGILITY* parameter specified by the application; the agility parameter specifies the time-window within which the application wishes the system to react to changes in the specified context, and, therefore, implicitly

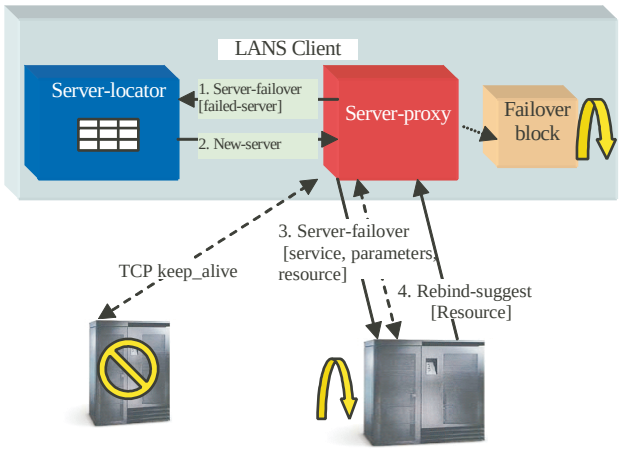


Figure 4: LANS Failover Protocol.

provides a measure of how quickly the system must be able to fail-over to an alternative server to hide the fault from the application.

Figure 4 sketches the LANS failover protocol. Failover is triggered when a LANS server fails to respond to a health-probe, causing the server-proxy to inform the server-locator about the failure. The server-locator purges the failed server from its records and returns another server from the same context, if one is available. If an alternative server is available, the server-proxy establishes a TCP keep_alive connection with it and sends it a SERVER-FAILOVER request, including in the request the description of the required service, the currently employed server, and the application-specified rebinding parameters. When invoked with a SERVER-FAILOVER request, the server starts a thread to discover, evaluate and compare the available alternatives, and may, at some later stage, suggest a rebinding if a better alternative becomes available. The SERVER-FAILOVER request however, unlike a routine SERVER-REQUEST, simply resumes the ongoing application-request at the new server by comparing the available alternatives with the currently used resource instead of starting anew by returning an n-best-list of matching resources.

If a server is not available for migrating the application request, the server-proxy triggers the failover-block in the client to handle the request locally. In this case, in addition to including the service-description, rebinding-parameters, and the currently used resource in the failover-request, the server-proxy also sends the socket_descriptor of the underlying socket to permit rebinding by the failover-block. In this case, the parser/worker module in the failover-block must parse the query in a constraint tree, and repeatedly invoke the resource-discovery and client-side evaluation modules to discover, evaluate and compare the suitability of available resources. If a better resource becomes available, the failover-block invokes the application-callback and triggers a rebinding of the application-session.

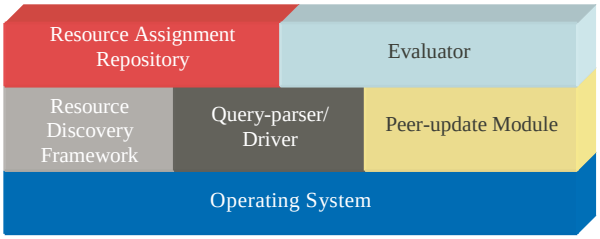


Figure 5: LANS Server Architecture

Importantly, recovery in LANS does not happen in the critical path of service-access. Applications continue to access the current service-provider while the failover happens in the background. Furthermore, given that application-supplied *agility* parameter is used as an upper-bound on how quickly the system must failover, the failover in LANS is designed to be transparent to the application.

IV.C. LANS Server Architecture

Figure 5 shows the LANS Server architecture. LANS server architecture extends SoNS's modular architecture by adding functionality for maintaining and disseminating the resource-to-application bindings for the "best-but-loaned" mode. Additionally, a LANS server can collate application-requests based on similarity in order to reduce the overhead of periodic discovery on available service-providers.

A LANS Sever is embedded with five modules:

1. The Query-parser/Diver module responsible for receiving an application request, parsing the service description in a constraint tree, and driving the whole system by periodically executing the adaptive discovery loop;
2. The resource discovery module defined by an extensible discovery framework, permitting various discovery protocols like SLP[10], INS, and SSDS[6] to be dynamically plugged in for discovering networked resources;
3. The evaluator module that assigns scores to the available resources based on their attribute values;
4. The resource-assignment-repository (RAP) that keeps track of the resources assignments to different applications, and finally;

5. The peer-update module responsible for disseminating information stored in RAP to other servers in the same context.

IV.C.1. Resource-reclaim

The LANS server architecture provides adaptive access much like the client failover block, except from one important addition to support the SO_NICE option. When requested to satisfy an application request with the SO_NICE option, a LANS server in fact satisfies the application request by proactively connecting it to the best possible resource—just as it would without the SO_NICE option. However, an application which sets the SO_NICE option could be subsequently requested to rebind to a less satisfactory resource to accommodate a new incoming request.

Though this optimistic approach resorts to a less-than-best resource allocation only when there is a conflict in application requests, it requires additional mechanism to dynamically re-allocate resources to satisfy a new request. To this end, the LANS server architecture includes two additional components: 1) the resource-assignment repository (RAP) to maintain a list of the resource-to-application bindings for all the application requests being handled in the SO_NICE mode, and 2) the peer-update module to periodically multicast the state stored in the RAP module to the RAP modules embedded in other servers in the same context. Given this information, when a server is invoked with a request, it also includes the "loaned" resources in the RAP module in its search for the top N matches for the application request. However, such matches are treated as a special case in two important ways: 1) Since a connection to "loaned" resource incurs an additional cost of rebinding an existing application to an alternative resource, matches from the RAP module are appended at the bottom of the list returned to the client, and 2) matches from the RAP module are returned with a special flag to indicate that a request to establish a connection with such a resource must go through the server, so that the server can interpose the required arbitration to handle the request. Furthermore, it is noteworthy that a request to establish connection with a "loaned" resource requires the server to find an alternative match for the victim application before it sends a RESOURCE-RECLAIM request to the respective client's server-proxy. In a naïve implementation, this would entail a new search for the victim application in the critical path of the incoming application request—effectively doubling the cost of handling an SO_NICE request. LANS avoids this cost by a simple, and almost co-incidental, optimization: since the LANS evaluation scheme is based on continuously evaluating and comparing the resources that match an application request, we store this periodically updated list of candidates as part of the RAP module. Subsequently, an incoming request that requires a resource-reclaim causes this pre-computed list of candidate choices (albeit the top choice which is to be taken away from the application) to be sent to the victim application in a RESOURCE-RECLAIM message, triggering a rebinding to accommodate the new application. Note that since the SO_NICE request is handled much like regular requests in the server-assist mode, in that better alternatives are sought opportunistically, the victim application could be upgraded to its top choice if it becomes available at a later stage.

IV.D. Collation of Application Requests

In order to reduce the overhead of periodic discovery at service-providers, a LANS server can amortizing the cost of discovery for similar requests. This optimization is based on the fact that an application-specific discover-evaluate-and-rebind cycle for, say, application A can in fact take advantage by a similar cycle for application B, as long as two conditions hold: 1) B's service request is a subset of the A's service request, and 2) A's probing frequency, *agility*, is less than that of B. Based on this observation, a LANS server collates requests as follows:

1. If an incoming request is a subset of an existing request, and its agility parameter is greater than the existing request, mark this request as a "subset-request"and use the n-best-list from the existing request to answer its periodic queries.
2. If an incoming request (A) is a subset of an existing request, but its agility parameter is more stringent than that of an existing request (B), start a "dummy-request", C, with B's service description and A's agility. Mark both A and B as "subset-request" of C.

V. Implementation and Evaluation

With its server-based optimizations, LANS is designed to reduce the overhead of proactive discovery and evaluation at a mobile client. Furthermore, since a LANS client can offload the task of network discovery and evaluation to a LANS server, the LANS architecture unhinges the performance of the system from the limited computation power available on a mobile

client. Below we present a series of experimental results to evaluate these claims.

Our evaluation is focused on two sets of experiments.

1. Comparison of the overhead of LANS and SoNS on a mobile client. We compare the overhead in power consumption, CPU usage, and network traffic.
2. Performance of the LANS server architecture for establishing and reconnecting the client connections.

The extended version of SoNS is implemented in Java so that it can be used across multiple platforms e.g. WinCE, Familiar Linux. For the purpose of comparison with the original version of SoNS (implemented in C), we also implemented a version of the LANS client in C for Familiar Linux. The C implementation includes the code required for basic resource discovery and evaluation using a LANS server; it does not include the code for fail-over. This was sufficient to permit a (language-independent) comparison of LANS and SoNS running on HP iPAQs.

V.A. Client Overhead

We compare the overhead of both systems using our running example, "offload-and-follow-me" video application. For this purpose, we setup our experiment on two HP H3600/H3700/H3800-series iPAQs with Familiar Linux 7.2-unstable, PCMCIA sleeves, and Orinoco Silver 802.11b wireless networking cards. We installed MPlayer 0.90, and both SoNS and LANS clients on the iPAQ. MPlayer was used to play a 320x176 MPEG version of the Matrix Reloaded movie trailer[1]. We chose this MPEG because it contains many action scenes, and as such, is a challenge for the iPAQ to decode. The iPAQ was connected to our lab network through a NATted Orinico RG-1100 Wireless Access Point. NATting the access point allowed us to control the traffic that the iPAQ saw and helped stabilize variation in our measurements. Our tests are the result of continual measurement over the duration of the 150s-long Matrix video clip.

We ran 5 tests:

1. Local Video with no adaptive service layer running.
2. Local Video with SoNS running in the background looking for better displays. The probe period was set to 1s and hysteresis was set to 3.

[1] Available from http://www.pocketmovies.net/detail_277.html

Figure 6: Our experimental setup for measuring the power consumed by an iPAQ and the wireless network card.

3. Local Video with LANS running in server-assist mode. We set the keep-alive probe to 60s.
4. Remote Video with SoNS running with the same parameters as above.
5. Remote Video with LANS running, with keep_alive set to 60s.

We set the number of available display devices to be between 5 and 15.

Local Video tests involved playing the trailer with MPlayer. Remote video tests stream off the video to an external display, with LANS or SoNS running in the background to look for better alternatives. For the local test, the iPAQ's LCD screen must be sufficiently lit up with backlight, consuming significant battery power. We shut off the backlight of the iPAQ's LCD when the video is streamed to a remote display.

V.A.1. Power Consumption

Since LCD backlight power consumption is constant for both SoNS and LANS, the key difference between the power consumption of the two is the usage of the wireless network card. SoNS uses OpenSLP[11], a peer-to-peer discovery model. It transmits a service discovery message every AGILITY number of seconds, and then collects the replies from matching resources until the next probe. As a consequence, the network card must be powered-up at all times, either sending or receiving data. Informal tests showed that enabling power-management on our network card for SoNS increased the latency to send and receive traffic by a factor of 10 to 30. On the other hand LANS in server-assist mode only relies on getting an occasional event from a LANS server, permitting the wireless network card to aggressively utilize its power-save mode. In our experiments we configure the power-save mode to be 0.1 (wake-up every 100 ms).

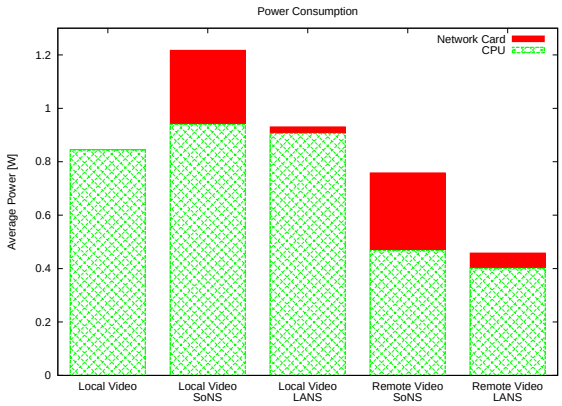


Figure 7: Power Consumption in Watts for the follow-me-video application.

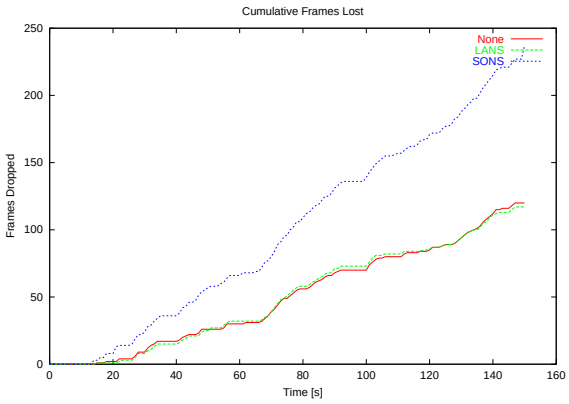


Figure 8: Cumulative number of dropped frames for the follow-me-video application.

We measured power consumption using a specially modified iPAQ and PC Card Sleeve. We removed the batteries from both so that an Agilent Variable Power Supply is the only power source. To measure total power consumption, we measured the voltage across and current though a precision resistor placed between the power supply and the iPAQ. Similarly, to measure network card power consumption we inserted the same precision resistor between the power supply and the network card using a Sycard PC card extender. Figure 6 shows the lab bench used for evaluation.

Figure 7 shows power consumption for each of the 5 tests. LANS does significantly better than the local-only version of SoNS. This is due mostly to the network card's ability to enter low power mode. Critically, LANS also makes offloading the iPAQs video to a bigger screen profitable from both a quality of service stand-point and a power consumption stand-point, while SoNS is only marginally better in the power consumption compared to the Local Video with no adaptive service.

V.A.2. CPU Usage

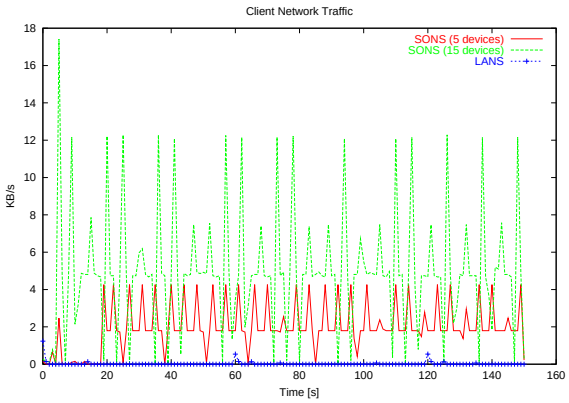
As mentioned in Section 2, SoNS consumed so much processor time that it interfered with the MPlayer on the iPAQ. Figure 8 shows the number of frames dropped when iPAQ is not running an adaptive service layer, when the iPAQ is running SoNS with local discovery, and when it is running LANS with a server framework. LANS offloads resource discovery to a server pool and, as a consequence, has a negligible effect on the performance of the video player. In the graph, steep slopes indicate a large number of dropped frames. Typically, this happens during actions scenes. SoNS tends to drop more frames for a longer period of time as compared to server-assisted LANS. Hence, our server-based optimizations, improve the quality of service (by streaming the video to a larger display) without adversely affecting the performance of the system when no external displays are available.

V.A.3. Network Traffic

Figure 9 shows a comparison of network traffic generated by both systems when the video is playing locally. SoNS deals with devices directly and generates significantly more network traffic than when it is in server mode. Worse, the overhead of network traffic for SoNS is proportional to the richness of the environment; if there are more devices, then SoNS must deal with each device individually. When using a server, however, LANS, generates a constant amount of traffic since the servers deal with the individual devices.

V.B. Performance of LANS Server Framework

The second set of experiments evaluate the performance of the LANS server architecture, measured on a Pentium III with 256MB of RAM running GNU/Linux 2.4. This is the same test bed as used in the SoNS evaluation in [13]. Figure 10(a) shows the time it takes to satisfy a connect() call from a LANS client using a LANS server as a function of the number of constraints in the service request. We use an in-memory resource discovery protocol to isolate LANS from network latency. Figure 10(b) shows the cost of rebinding an existing connection; the time elapsed from when a call to *run_query()* method returns—with matching resources on the network—and when LANS invokes the application-callback to notify the presence of a better alternative (given the hysteresis semantics). This latency again increases only linearly with



	RX Bytes	TX Bytes
SoNS (5 Devices)	150829	152094
SoNS (15 Devices)	406148	353182
LANS	2037	1395

Figure 9: Network traffic for the follow-me-video application.

the number of constraints (both for nested and non-nested constraints). Even with its Java-based server-side implementation, LANS performance is comparable to SoNS and more importantly, is unhinged from the limited processors in a mobile client.

VI. Related Work

LANS is designed to leverage dynamic resource discovery protocols to provide opportunistic access. As opposed to the white-pages style lookup offered by systems like DNS that simply resolve a resource name to its network address, dynamic resource discovery systems do not require a priori knowledge of some unique identifier of the resource, like its network address, and hence can be used to dynamically discover and utilize resources as they become available in a pervasive system. In addition to the classical examples like Grapevine[1] and X.500 [2], a range of industrial standards like Microsoft' UPnP resource discovery protocol (SSDP) [8], IBM's T-Spaces, and IETF's Service Location Protocol [10], and experimental systems like MIT's INS and Berkeley's SSDS [6] have emerged over the last few years. LANS, like SoNS, is designed such that different discovery protocols can be added to its resource discovery module, possibly via a simple wrapper function to covert the SoNS attribute list to the specific format used by a discovery protocol, e.g. XML (used by Berkeley' SSDS).

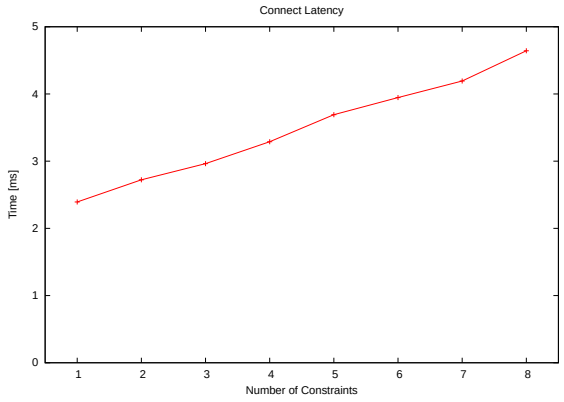
SoNS provides an alternative to content-based routing systems like INS' late binding architecture and the Information Bus [3] which provide adaptive service access by permitting an application to send messages without specifying the network address of the recipient, and route the messages to the appropriate server by looking at the content of each network message and matching that with the properties of the available servers. Where such systems offer an alternative to our approach, they inherently lack application-level session semantics, do not offer a clean interface for configuring the application-specific policy for resource comparison and session-rebinding, introduce the overhead of resolving service descriptions within the critical path of every message delivery, and, by defining their own routing framework, do not leverage the support for QoS offered by the underlying network.

SoNS's use of a server framework for shedding load from a resource-constrained mobile device is comparable to traditional proxy-based mobile architecture. However, the SoNS client-server architecture is unique in its attention to robustness and scalability, and in its special treatment of network-sensitive vs. resource-specific attributes in a service request.

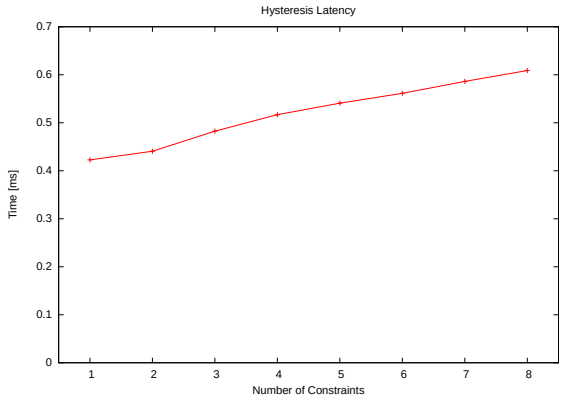
As opposed to the previous related systems that match a service request with the "best" possible available resource, SoNS introduces the concept of utility-based resource selection and rebinding for opportunistic service access. This approach is influenced by economy-based resource sharing systems, as well as AI-based reasoning systems like [9]. However, while systems like MetaGlue [9] propose to use general-purpose constraint satisfaction engines over complex utility functions, SoNS evaluator is designed to be simple and responsive to changes in the system.

VII. Conclusion

In this paper we report on our experience with two different systems for offering adaptive service access, Service-oriented network sockets, and the Intentional Naming System. We conclude that though the session-oriented semantics offered by SoNS are imperative for most pervasive application, its purely end-to-end architecture results in an unacceptable computation and communication overhead at resource-



(a) Latency for a connect() socket call.



(b) Latency incurred on rebinding.

Figure 10: Performance of the SoNS server framework.

constrained mobile clients. Furthermore, the greedy "best" match policy offered SoNS and INS typically leads to a poor utilization of resources in shared spaces. Finally, the simple semantics of resource rebinding implemented by SoNS and INS, often lead to perfunctory reconnections, which may be counterproductive for the application function.

Based on our experience, we present the design and implementation of Lightweight Adaptive Network Sockets (LANS) for accessing services in a dynamically changing networked environment. LANS is an improvement over previous systems in three key dimensions: 1) The LANS session layer offloads the process of discovering, monitoring, evaluating and comparing the available resources to nearby LANS servers, reducing the computation and communication load on a mobile client while still maintaining session semantics. 2) LANS is designed to offer richer semantics of resource selection and allocation to maximize the utility of available resources; in addition to a simple "best" match, LANS supports "barely-sufficient" matching, as well as dynamic reallocation of resources to satisfy competing applications in a shared space. 3) LANS permits an application to control the opportunistic behavior of the system by giving applications increased control over when they are notified about better resources. Our experimental results show that, with its server-based optimizations, LANS achieves better performance, consumes less battery, and has lower network overhead than SoNS.

References

[1] Andrew D. Birrell, Roy Levin, Roger M. Needham, and Michael D. Schroeder. Grapevine: An exercise in distributed computing. *Comm. ACM*, 24(4):260–274, April 1982.

[2] CCITT. The directory—overview of concepts, models and services,, December 1988. X.500 series recommendations, Geneva, Switzerland.

[3] B. Oki et al. The information bus (r) - an architecture for extensible distributed systems. In *Proc. ACM SOSP*, pages 58–78, 1993.

[4] Christopher K. Hess et. al. Building applications for ubiquitous computing environments. In *International Conference on Pervasive Computing (Pervasive 2002)*, pages 16–29, Zurich, Switzerland, August 2002.

[5] M. Esler et. al. Next century challenges: Data-centric networking for invisible computing: The portolano project at the university of washington. In *Proceedings of Mobicom 99*, 1999.

[6] S. Czerwinski et al. An architecture for a secure service discovery service. In *Proc. of MobiCom-99*, pages 24–35, August 1999.

[7] William Adjie-Winoto et al. The design and implementation of an intentional naming system. In *Proc. 17th ACM SOSP*, Kiawah Island, SC, December 1999.

[8] UPnP Forum. Universal plug and play. http://www.upnp.org.

[9] Krzysztof Gajos. Rascal—a resource manager for multi agent systems in smart spaces. In *Proceedings of CEEMAS 2001*, 2001.

[10] Erik Guttman. Service location protocol: Automatic discovery of ip network services. *IEEE Internet Computing Journal*, 3(4), 1999.

[11] Openslp. http://www.openslp.org/.

[12] Project oxygen. http://oxygen.csail.mit.edu.

[13] Umar Saif and Justin Mazzola Paluska. Service-oriented network sockets. In *Proceedings of the USENIX Annual Conference on Mobile Systems, Applications, and Services: MobiSys 2003*, 2003.

[14] J.H. Saltzer, D.P. Reed, and D.D. Clark. End-to-end arguments in system design. *ACM Transactions in Computer Systems*, 2(4):277–288, November 1984.

[15] M. Tuexen, Q. Xie, R. Stewart, M. Shore, L. Ong, J. Loughney, and M. Stillman. Architecture for reliable server pooling. RFC3237, January 2002.