Suspending and Resuming Network Applications Using Session Continuations

Alex C. Snoeren, Debashis Panigrahi, Shoubhik Mukhopadhyay,
Hari Balakrishnan, and M. Frans Kaashoek

Abstract

Migrate is a system service that allows network applications to suspend upon disconnection and properly resume when connectivity is restored. Migrate uses *session continuations*, which allow applications to explicitly record all the state and resources required to correctly resume. Migrate virtualizes network connections to function across network address changes, allows portions of an application to be suspended and securely resumed, and optimizes scarce system resources on servers handling many suspended clients. We demonstrate that the continuation abstraction is both useful and efficient by showing how two important applications, SSH and Icecast, can use Migrate to implement a suspend/resume feature. We find that the required source code changes are between 0.5% and 1.75% of the total, that the generated continuations are between 1% and 5% of the memory footprint of the entire process, and that session resumption times are small (a few hundred milliseconds) compared to typical disconnection durations. We also show that he proposed system is scalable to support large number of simultaneous requests for suspension/resumption.

I. INTRODUCTION

Recent advances in wireless networking technologies are finally bringing the vision of ubiquitous, pervasive mobile computing to today's Internet users. The proliferation of 802.11-based Wi-Fi hot spots and 3G wireless broadband deployments has enabled users in many major metropolitan areas to access the Internet almost anywhere using their personal mobile devices. Additionally, developments in virtual machine (VM) technologies have renewed interest in anonymous, public workstations that serve as a portal into a user's portable operating environment. These disparate but complimentary technologies both enable a common paradigm of interaction, which we call Internet suspend/resume.

It is common for people to use their laptops and handheld devices from several different networks, often changing networks several times every day. Additionally, in a distributed service infrastructure, it is often required to transfer application service from one server to another for balancing the load as well as providing better service from servers at proximity.

While much previous work has focused on protocols for preserving network connectivity as users *continuously* move within and between networks [13], [15], [19], [23], [27], very little system support has been developed to preserve application-level communication for the more common problem of hosts disconnecting from one network and resuming *later* from a different network. Such support is crucial for the seamless operation of applications that establish *sessions*,

or long-lived connection-oriented associations between multiple end points. For these applications, which can include Web sessions, file transfers, streaming media, and remote interactive shells, application state is distributed between the end points. This paper describes Migrate, a system that enables network applications to *suspend* sessions upon disconnection and correctly *resume* when connectivity is restored.

The challenge in the design of a network suspend/resume service is how to manage and restore distributed state. An approach that transparently packages up the entire state of each communicating process upon suspension (e.g., through check-pointing) and re-instantiates it upon resumption is not a full solution to the problem for several reasons. First, today's connection-oriented transport protocols like TCP name connection end-points by their IP addresses, so transparent process resumption at a different network location requires some form of connection virtualization [23], [24], [27] that allows communication across IP address changes. Second, even with such virtualization, an approach that transparently suspends and resumes entire processes does not work in the case of a single process (*e.g.*, a Web server) serving multiple connections, in which client connections may suspend independently. What is needed is the ability to suspend and resume a portion of a process in this case. Third, an unoptimized suspend/resume method can consume a disproportionately large amount of scarce system resources in the form of system memory, file descriptors, and network bandwidth. Finally, exposing disconnection to the application allows it to specify different methods of resuming, including terminating current application processes and spawning new ones upon resumption.

We propose migrate, a system service that solves these problems by using a new abstraction called a *session continuation*. Session continuations are inspired by the continuation passing style (CPS) in programming languages [1], [20]. In CPS, a procedure terminates by calling another and therefore does not need to return, which in turn simplifies compiler construction by easing state management. In the networking context, an application generates a session continuation that records all of the state and system resources needed to process the "rest of the session." Upon resumption, the system simply invokes the suspended session's continuation. Because a session continuation contains all of the information required to correctly resume a suspended application, the system can safely reuse any resources previously allocated to suspended sessions.

We have implemented Migrate in Linux as a user-level daemon and support library. Session continuations allow applications to save the subset of their user-level and system state needed for

resumption, for which they use the library's continuation API. This subset is often significantly smaller than the full process state, making the abstraction useful in practice. Our design of continuations allows sessions to be resumed either within the context of the original hosting process, or as new processes. It also allows continuations to be generated either as soon as disconnection is detected, or more lazily. The Migrate daemon manages these continuations, optimizes the system resource consumption of suspended sessions (*e.g.*, file descriptors and kernel memory), and monitors end-to-end connectivity on behalf of sessions to determine when to suspend sessions and securely resume them.

To demonstrate the effectiveness of the proposed system to suspend/resume network applications using session continuations, we have integrated them into two popular session-based Internet services: the Secure SHell (SSH) [26] and the Icecast streaming media server [8]. These applications present a number of contrasting application design points, such as event *vs.* thread-based, one *vs.* many clients per process, etc. We show that session continuations are flexible enough to accommodate both in a straight-forward fashion, and encapsulate application state and almost all the resource dependencies in a generic fashion. The source code change required is small, between 0.5% (SSH) and 1.75% (Icecast) of the total code. The generated continuations are only between 1% (SSH) and 5% (Icecast) of the memory consumed by these applications.

We present performance numbers from our Linux-based implementation that show session continuation based suspend/resume service outperform VM-based approaches by an order of magnitude in the (limited) cases when VM approaches are viable. This time is small compared to typical disconnection durations. We also find that the throughput and latency impact of transport protocols using Migrate on the data path is small for real-world networks, and that this overhead can be eliminated using an in-kernel TCP migration protocol [23] if desired. Finally, we show that the system is scalable to handle a large number of simultaneous requests for suspension/resumption.

The remainder of this paper is organized as follows. We begin in Section II with an overview of the scenarios where Migrate service of application suspension/resumption can be used and present an overview of the service. Section III introduces the concept of session continuations and their use in application servers. Section IV presents the system architecture to enable suspend/resume service using session continuation and migration service as building blocks. We describe the implementation of the proposed system service in Section V. Section VI presents a quantitative

evaluation of the implementation complexity, migration performance and scalability. We then compare our work to previous approaches in Section VII before concluding in Section VIII.

II. System Overview

In this section, we present an overall architecture of our session migration system and highlight different issues associated with it. First, we motivate the use of suspend/resume service through a few example usage scenarios. Next, we present an overview of session layer abstraction that plays an important role in the service. Finally, we describe the steps involved in enabling the suspend/resume service for sessions.

A. Example Use of Suspend/Resume Service

Figure 1 presents different scenarios where the suspension/resumption of application can be useful. In the base scenario, a client is initially accessing a network application service at `Server Location 1` from `Client Location 1`. The server at location 1 can potentially interact with other helper services, such as X-display, or any external node. There are several potential scenarios which necessitate suspending the application and resuming at the same location later in time or at a different location (`Server Location 2`). One of the potential scenarios is to preserve application state through network disconnection. Upon network unavailability, application server is suspended and resumed later when the network is available. The second scenario is caused by client mobility, as the client moves out of a given network to another, the session can be suspended after disconnection and resumed after connectivity is restored in the same network or a different network. Additionally, as the client moves to a new location the session can potentially be better served from a different replica server than the current one necessitating the migration of server. Finally, a scenario where the server migration would be helpful is in load-balancing between multiple servers handling many different client connections.

Driven by the needs of the above scenarios, the goal of the migration system is to suspend the service as required and transfer the application service to the same server or another replica server along with any associated sessions between the server and external resources, and allow the client to continue accessing the service at the new server without going through the service setup all over again. The application service may have some interaction with external nodes or use local helper services. The similar interaction needs to be resumed at the destination location.

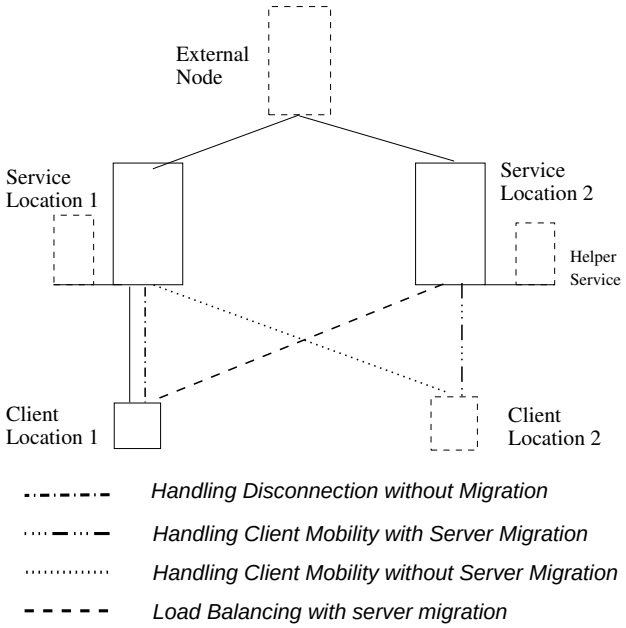


Fig. 1. Example Use of Suspend/Resume Service

Apart from providing application service availability for the scenarios illustrated above, namely network disconnection, client mobility, and server load balancing, another major advantage of application suspension/resumption is conservation of scarce resources. During period of suspension, system resources used by application services, such as file descriptors, memory, buffers, can be conserved. Additionally, the application can lead to conservation of other resources indirectly, such as energy, by shutting down the network interfaces during periods of disconnection.

A system providing server migration has the following needs: (1) a mechanism to handle network mobility of client/server and (2) a means of preserving application state for the service to resume later. For handling network mobility, we leverage Migrate's session abstraction which we briefly discuss below. For application state preservation we use *session continuations*, an abstraction that enables servers to manage the suspension and resumption of sessions at a different location.

B. Session Abstraction

Migrate introduces a session layer to the Internet protocol stack to assist applications in dealing with mobility. A session is a collection of transport-layer connections. While a session can in general encompass any number of end points, in this work, we focus on sessions between exactly two end points.

With Migrate, connections in a session survive periods of disconnection and changes in attachment point while preserving the semantics of the selected transport protocol. In other words, one or both of the client and the server can change network attachment point (*i.e.*, move to a different IP address) or disconnect at any time without adversely affecting the session. Communication will resume as soon as connectivity is restored. Migrate can preserve TCP connections in two ways: it uses the TCP Migrate options [23] if available, or falls back to a user-layer virtualization technique that synthesizes a logical connection out of multiple physical ones [22].

Migrate allows applications to name session end points using any naming system of their choice, such as the Internet DNS. Applications provide Migrate with the names of the end points and a method to resolve them. Migrate sessions track the end points as they change attachment points, maintaining end-to-end communication between end points. If either end point changes attachment point or disconnects for some period of time, it securely notifies the other end point of the change when connectivity resumes. If both end points move at the same time, Migrate uses the name resolution method provided by the end points to resume communication. The APIs for using the Migrate functionality is described in detail in the Appendix **??**.

C. Suspend/Resume System Overview

Within the scope of the Migrate architecture, we implement migration functionality broadly in four steps: (1) suspending the current session and generating a session continuation, (2) validating the feasibility of session continuation upon resumption request, (3) transferring the continuation with associated application state and system resources securely to the new server if needed, and (4) invoking the continuation at the new server to resume the application. The steps involved in application session suspension/migration is shown in Figure 2.

In our system, we assume that the suspension of the original session may be initiated by either of its end points, triggered by application events or network connectivity events. Upon disconnection notification from network (shown as Step 1), Migrate informs application to suspend the service. As a part of the suspension process, the system generates a *session continuation* for the application. In addition to preserving necessary state and control function, generation of continuation needs to handle all currently open system resources and take appropriate steps to make it available later at resumption.

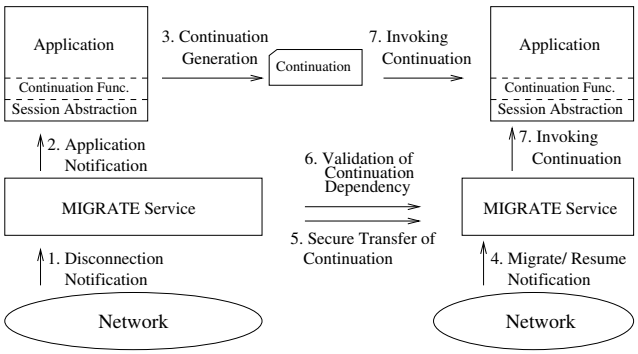


Fig. 2. Steps in Application Suspension/Resumption

At the time of resumption, the migration/resumption notification is received by the destination Migrate service node. For a simple implementation without sacrificing generality, we assume that the request to begin application migration/resumption, *i.e.* to transfer and execute the session continuation, needs to originate at the client host only. Upon resumption notification, the Migrate services on the source and destination nodes coordinate to evaluate the feasibility of invoking session continuation at the destination by validating the various system/application dependencies specified in the continuation. After successful dependency check, the session continuation is securely transfered to the destination node (Step 6). Finally, the session continuation is invoked to resume the application at the destination node.

Having presented the overview of Migrate-based suspend/resume service for applications, next, we present in detail *Session Continuation*, backbone of our system.

III. Session continuations

By making the notion of the "rest of the session" explicit, session continuations enable graceful handling of session disconnection, reconnection, and re-binding. Upon disconnection, server sessions generate a continuation that specifies in a generic fashion how to resume the session and includes any state and information about the necessary system resources. Upon reconnection, the server invokes the safely stored continuation to resume processing. Unlike a process checkpoint, a session continuation is *not* simply a snapshot of the current local state. A state snapshot is problematic to capture and restore and is likely to be inconsistent with current conditions at the time and location the continuation is invoked. Rather, a session continuation is a function from the state of the end points at session resumption time to an execution context, such that returning control to the application with that context continues the session correctly.

In most cases, the state that must be preserved in a continuation is substantially smaller than the original state and set of resources; a continuation may provide a session with alternate, equivalent state and resources. For example, in some cases Migrate may replace the original application process with a new copy. In other cases, changes in end point or network conditions may dictate that the application state upon resumption be different, reflecting the new network attachment point as well as current network and end point characteristics (*e.g.*, link bandwidth). The relatively small size of session continuations as compared to process or virtual machine checkpoints [18], [21] is a significant advantage for resource constrained hosts or busy Internet servers where scalability is important.

Despite their disparate structures, a common property of network applications is that they function as event loops, iteratively processing client events, blocking only on external I/O operations. As a result, it is usually possible to find a point in the control flow of the application, otherwise called quiescent points, where the state of a session can be described concisely. The more straightforward this description, the simpler the continuation generation. Finding such a quiescent point to generate continuation is also simplified by the ability of network applications to execute in the absence of network connectivity. Hence, it is typically straightforward to identify quiescent points where the suspension and resumption could occur. In particular, after completing all pending tasks, network applications use the `select()` system call to block awaiting additional input, so generating a continuation at this point is straightforward; in fact, this is when the application generates the continuation regardless of when the daemon notifies the handler of disconnection. It is important to note that the decision of quiescent point should ensure that the point can be reached from any point in the application after network connectivity is lost; there by guarantees generation of appropriate session continuation.

Session continuations fall naturally into three classes, corresponding to the architecture of the particular application:

- **System continuations (**C_{sys}**)** continue portions of the session managed by the system, such as network connections, file handles, and so on. Such continuations can be generated and executed by the system transparently to applications.
- **Internal continuations (**C_{int}**)** continue a session within its hosting process. For applications like Web servers that service many sessions at once, the disconnection of one session generally does not lead to the suspension of the entire application.

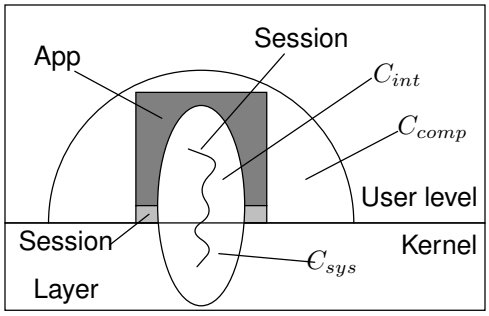


Fig. 3. Three separate continuations make up a complete session continuation: a base continuation, C_{sys}, an internal continuation C_{int}, and one that restarts the entire application, C_{comp}.

- **Complete continuations (C_{comp})** continue a session in the absence of its original process (*e.g.*, by starting another instance of the same application). Many applications (SSH and FTP, for example) start a new process for each session which can be discarded if the session suspends.

If the application state does not change during disconnection, the continuation simply needs to restore the system resources required by the session and identify the point in the application to which control should be returned. We call this simple preservation continuation the system continuation, C_{sys}. A special category of system continuation that preserve system resources is *resource* continuations that effectively restore the resources when called, but release them for use by other applications until then. One example of a class of system resources that can be effectively conserved while a session is disconnected is file descriptors. Because Migrate manages the network connections associated with open communication sessions, it is easy to generate continuations that release file descriptors for suspended network connections by aborting the connection and instantiating a new one upon resumption. Doing so can realize considerable resource savings in practice. In addition to the file descriptor data structure itself, network sockets have associated send and receive buffers containing data that has not yet been sent by the operating system or received by the application, respectively. These buffers need to be large for good performance over high-speed links. Buffer memory in contemporary operating systems is fixed or unpageable, meaning that a suspended session's connection buffer may decrease the amount of *physical* memory available for active sessions. Migrate also conserves swap space by freeing application memory dedicated to session state and storing a compressed version of the attribute/value store as part of each session's continuation.

While C_{sys} preserve the state of the system resources, application state associated with a

session might change through the period of disconnection. Hence, in addition to system resources, application state needs to be preserved with state information distributed between the client and the sever. This is an example of an *internal continuation*, C_{int}, which needs to contain sufficient application state to allow the session to continue including a function that restores the context required by the session. Internal continuations are useful for applications like Web servers that service many sessions in one process, where the disconnection of one session generally does not lead to the suspension of the entire application process.

Depending on the application design, if session resumption necessitates starting of appropriate application process, in addition to preserving system resource and application state, the context of the application needs to be preserved to be resumed appropriately in future. This type of continuation is called C_{comp}. Many applications including SSH and FTP start a new process for each session, which can be discarded if the session suspends. A complete continuation resumes the application process through its own continuation, C_{comp}, before returning control to it. A complete continuation is a composition of continuations as shown in Figure 3. The contents of C_{comp} depend on the operating system. In our UNIX implementation it includes the process' command line arguments, environment variables, and the current directory. The more general the representation, the more portable the continuation: in fact, a complete continuation could be run on a system different from the one that originated the session.

Another base of continuation classification is the location of session resumption. The definition of each of the above type of continuation depends on whether the session is resumed at the same host where the continuation is generated or at a separate host. While system resources are made available by Migrate service, appropriate session resumption steps need to specified by the application to resume service across hosts using the continuation generated for resumption at a separate host.

The type declaration of a Migrate session continuation is shown in Figure 4. This structure is composed of three classes of information. The first represents C_{sys}: a set of file descriptors corresponding to files, devices, UNIX pipes, etc. that will be needed upon resumption, in addition to the network sockets that make up the session. File descriptors and sockets included here are no longer available to the application and will be restored only upon execution of the continuation. The second component of the continuation structure is the internal continuation function, C_{int}, defined within the application process that Migrate should call upon resumption of connectiv-

```
typedef struct {
  fd_set          fds;      /* IPC to preserve: C_∅   */
  void *          db;       /* Data to preserve       */
  cont_func       cont;     /* Cont. function: C_int  */
  const char *    argv[];   /* Cmdline args: C_app    */
  const char *    envp[];   /* Env. variables: C_app  */
  const char *    cwd;      /* Current dir.: C_app    */
} migrate_continuation;
```

Fig. 4. A Migrate continuation structure contains a set of file descriptors that must be preserved, an attribute/value store, and the continuation function itself. Complete continuations also specify several parameters used when restarting the application process.

ity (instead of the normal mobility handler specified through `register_handler()`). The third class of information constitutes C_{comp} which includes command-line arguments ($argv$), environment variables ($envp$) and the current working directory (cwd).

By generating a session continuation, applications are able to simultaneously manage the suspension and resumption of disconnected sessions while allowing the host operating system to reclaim application and system resources during disconnection. Each Migrate-aware application can suspend a session by providing its own session continuation. Migrate reclaims system resources by additionally generating incremental resource continuations that are composed with application-provided continuations. Section VI evaluates the effectiveness of Migrate's resource continuations with respect to system memory, open file descriptors (both those associated with network connections and otherwise), and power consumption.

IV. Suspend/Resume using Session Continuations: System Level Issues

The basic steps involved in the proposed Suspend/Resume system architecture were introduced in Section II. In this section, we highlight the issues involved in enabling the service as presented before and present the solutions of the issues as provided in the Migrate system.

A. Connectivity Monitoring/Application Notification

One of the important issues in Migrate service of suspension/resumption is to monitor connectivity and notify the application accordingly to suspend (generate continuation) or resume

(invoke continuation). Depending on the needs of the applications, the decision to migrate the server could be made by any one of the various agents involved: the client, the initial server, the final server, or most likely by some external system modules that can monitor the current network attachment point and/or server activities. Upon change in network attachment point, the monitor agent can interact with the system to evaluate the feasibility of migrating the server to potential new locations, and select a location guided by pre-assigned service policies.

To decide if an end point is connected at its current attachment point, Migrate needs to monitor the connectivity of on-going sessions. Unlike existing approaches that assume connectivity is strictly a function of the local attachment point [9], Migrate's notion of "connected" can vary from session to session. To be connected, application sessions may require a certain level of connectivity, expressed in terms of available bandwidth, maximum latency, or similar metric. Hence, Migrate needs a suite of modular *connectivity monitors* to assist in evaluating the current levels of connectivity being experienced by each session. Connectivity monitors may gather information from a variety of sources, including the physical and network layers (*e.g.*, loss of carrier, power loss, change of address, etc.), the end point applications themselves, or appropriately authorized external entities (*e.g.*, [2]) that may be concurrently monitoring connection state. Since a session may span multiple protocols, connections, and application processes, there may be several sources of connectivity information for any particular session.

Using the connectivity information gathered from different sources, migrate needs to decide on the appropriate action to be performed including the location of target servers for resumption.

B. Session Continuation Generation/Invocation

Based on the notification generated by Migrate, the next issue is to generate appropriate type of continuation at the desired time and select desired continuation type at the resumption. The generation/invocation of continuation depends on the application architecture and application-level policies.

When Migrate notifies an application upon detecting disconnection, the handler can generate session continuations either *eagerly* or *lazily*, depending on the application. Some applications require immediate notification of disconnection, usually to buffer outstanding data, claim or release locks, or reset timers. Alternatively, the application can execute unaffected during disconnection and generate the continuation lazily. The continuation is not generated until the system needs to

reclaim the resources currently being used by the disconnected session.

Another application level policy related to session continuation is to decide on the type of continuation to generate as it might affect the feasibility of resumption later. For example, if a complete continuation is generated assuming resumption at the same host, the application service can not be resumed. Similarly, if C_{int} generated is intended for resumption at the same server, the session can not be resumed at a different host..

Apart from application-level policies to decide generation of appropriate continuation, the application instrumentation necessary for continuation generation depends on the application architecture. For example, finding quiescent point for an application process is very different from threaded application. In case of threaded application, in addition to finding the information to be stored in the continuation, the place of continuation generation is important and has to consider possibility of any deadlock because of data dependency across threads. Additionally, for a threaded application, for communication between migrate service and application a master controlling thread has to be identified for notification/information exchange.

C. Session Continuation Management

While session continuations enable resource savings, they incur costs as well. In particular, each individual session continuation must be stored and managed by both disconnected end points until connectivity is restored. In some cases, however, connectivity may never be restored; two end points may never come in contact with each other again.

Hence, session continuations must be *garbage collected* at some point—continuations that are deemed useless need to be purged from the system. Garbage collection creates two complications: First, at what point is it appropriate to discard an apparently unwanted continuation? Second, since the application that created the continuation expected it to be invoked, how can the continuation be disposed of while ensuring the application is not left in an inappropriate state. The first issue is quite complicated and remains an area for further study. Migrate currently enforces an expiration date on session continuations established through a combination of application, user, and system-wide policy.

D. Session Dependency

A server application can potentially have multiple sessions which are dependent on each other. An example case of session dependency is shown in Figure 5. For instance, an SSH server that provides X forwarding would have (at least) two sessions per client: one containing the connection between the SSH client and SSHD server (S1), and additional sessions between the SSHD server and each open X application (S2). (Recall that X applications communicate with the X server through TCP connections.)

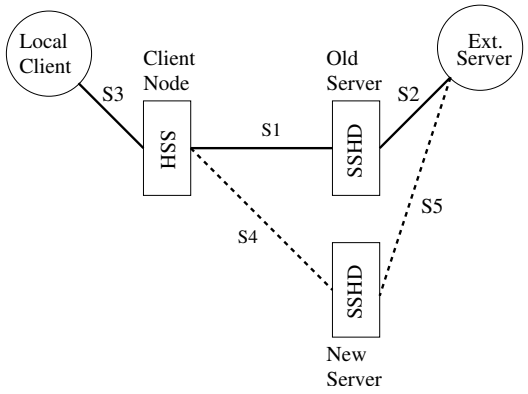


Fig. 5. An instance of session dependencies for SSHD server where S3 is dependent on S2.

In these instances, we must take appropriate steps to ensure the complete migration of all sessions constituting the service when a client changes servers, honoring any dependencies between them. For example, as shown in the Figure 5 when the session S1 is suspended, the dependent session S2 also has to be suspended. Upon migration of SSHD from old server to new one, as S1 is migrated to become S4, one end of the S2 has also to migrate (to S5). In order to take appropriate actions, it is important to understand the dependency between different sessions of an application. In most cases, the dependency between sessions is one-directional, *i.e.*for a pair of dependent sessions, suspending the first should result in suspension of the second, but not vice versa.

It is practically infeasible to implicitly estimate inherent dependency between different session automatically. Our current implementation requires server applications to explicitly provide information about the dependencies between different sessions belonging to the application, but we are considering techniques to automatically detect such dependencies.

E. Server Dependency Verification

In order to enable applications to effectively use session suspend/resume, there is a need to verify the compatibility of the execution environment where the session is resumed compared to the one where it was suspended. This is especially important for long-lived applications which maybe resumed after a long time or those which need to move a suspended session to a different before resuming. The parameters that need to be compared may include environment variables, available versions of supporting programs and libraries. This may depend on several factors including future resource requirements of the application, system settings and service configuration at the new host. Some of these properties are generic enough to be verified by migrate, while some properties are specific to the service and need support from the application. Examples of such configuration parameters include the version of the service available, the existence of a particular library, availability of specific NFS mount points, default configuration of the service (such as X-forwarding option for sshd).

Apart from the issues mentioned above, there are other issues related to the system, such as authentication, session synchronization, which would be discussed later. Having described the different components of the suspend/resume system using session continuation, in the next section, we will present salient features of our implementation.

V. IMPLEMENTATION

We have implemented the Migrate sservice in Figure 6 as a user-level daemon in Linux. Our implementation of the session layer is a dynamically loadable library linked against individual applications. The main support infrastructure for the session abstraction is implemented in the Migrate daemon, which interacts with both local policy engines and with a suite of connectivity monitors to manage open sessions.

Here we first describe the interface presented to the application by the session layer library and illustrate how the interface can be used to access the suspend/resume services. This will be followed by a discussion of the code instrumentation necessary to use these APIs for some example applications. We shall also describe the implementation of the Migrate daemon followed by the control protocol used between different hosts during session migration.

A. Session Continuation Interface

Table I lists the extensions to the Migrate API that allow applications to construct and manipulate session continuations. A key feature of session continuations is the ability to preserve session state separately from the hosting applications. To simplify the task of continuation generation for applications that lack their own mechanisms for maintaining session state persistently, Migrate provides a simple persistent attribute/value store. Applications add continuation state to the store using the `store()` call. This state is then available through the `retrieve()` function until the session is destroyed either using `session_close()` (Appendix **??**) or by the continuation garbage collector (Section IV-C). Applications that need to know how large the value of an attribute is before retrieving it from the store (to allocate a sufficiently large buffer, for example) can call `store_size()`.

```
int store(Session s, char *attr, void *value, int len)
int retrieve(Session s, char *attr, void *buffer)
int store_size(Session s, char *attr)
int export(Session s, char *attr)
int freeze(Session s)
int return_cont(Session s, Continuation c)
int register_handler(Session s, Handler h)
int register_attach_handler(Handler h, int ServId)
int register_compat_vector
                (Session s, char *attr, void *val)
int compat_store(Session s, char *attr, void *value)
```

TABLE I

SESSION CONTINUATION API.

In certain instances, sessions build up shared state that is not readily observable from the remote end point, but may be necessary for session resumption. Rather than require the application to explicitly communicate this state as part of its continuation, Migrate supports the exchange of small amounts of state during session resumption. Applications can request the exchange of any attribute in a session's store through the `export()` call for use by the remote continuation. The session control protocol communicates exported values to the remote end point upon reconnection.

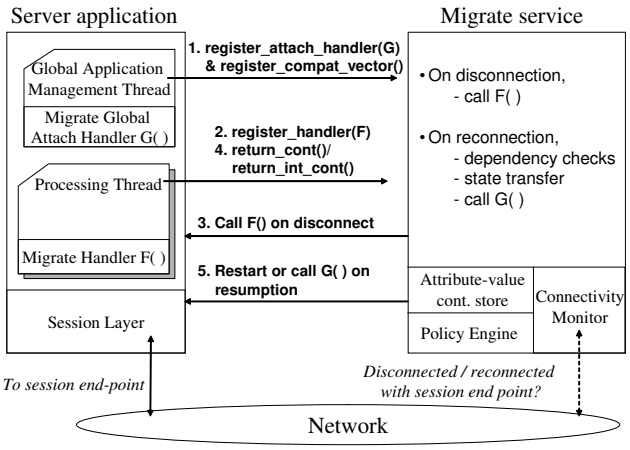


Fig. 6. Usage of Migrate's session continuation API to provide suspend-resume support in a multi-threaded server.

For each session, the application can setup a call-back function to handle disconnection notifications from Migrate through the `register_handler()` function. The call-back function registered is called by Migrate to trigger a continuation generation. For multi-threaded applications which contain a global monitor thread handling new server sessions, an attach handler may be set up using the `register_attach_handler()` call. This call-back would be called on resume to invoke a continuation and attach the resumed session to the already running service.

To support server dependency verification (Section IV-E), Migrate maintains a registry of compatibility parameters for all applications that require migration support. The registry is updated through the `register_compat_vector()` anytime a Migrate-aware service is installed. During continuation generation, the application can use the `compat_store()` API to export a vector of compatibility tags to the Migrate service, expressing the set of server-related dependencies that need to be satisfied to run the service. Each tag, of the form `<id,name,value>`, carries an `id` specific to a property, name of the property and the value to be checked. Before starting the process of migration, the property checking is performed in a step that uses the dependency tags exported by the application and the destination host's registry to check whether they are compatible.

Figure 6 shows how the APIs can be used to add support for continuation-based session migration to a multi-threaded server application, along with the variations necessary for single process applications. At startup, applications register the global attach handler (G) and compatibility vectors(step 1). As sessions are started up, the thread handling each session

also registers the migrate handlers (F) for each session with the Migrate service (step 2). For a single process application, there will be no global handler, and the compatibility vector registration would be done in step 2. When the connectivity monitors detect a disconnection for a session, Migrate invokes the registered migrate handler (step 3). The handler generates a session continuation (step 4), which could be a *complete* continuation (for process-based servers) or an *internal* one (for multi-threaded servers). After storing the continuation with Migrate the application process/thread exits, discarding the state and freeing up the resources associated with the suspended session.

For separate host migration, migrate now decides on the new host to migrate the suspended session to, and transfers the continuation and associated state to the new location after verifying the authenticity of the new host and the compatibility of the suspended state with the new environment. The compatibility verification step in not required for same host migration. When the connectivity monitor subsequently determines that connectivity has been restored for a disconnected session, Migrate invokes the stored session continuation, which causes the application to resume the session from the point specified in the continuation (step 5). For single process applications, this is done by restarting the application which executes the complete continuation at startup. For multi-threaded applications, upon execution of the global handler by migrate service on the new host the global application management thread invokes the continuation to attach a new session. Finally, Migrate may call `freeze()` at any time on a suspended session that does not have a stored continuation if it wants to reclaim system resources (*e.g.*, file descriptors or memory) that are tied up by the application. In addition, on each disconnection or reconnection, the Migrate service consults the policy engine (Section V-E) to decide which of several possible network interfaces is currently best suited for each session.

B. Application Instrumentation

To understand the instrumentation steps necessary to enable application migration, we first discuss different parts of code modification using examples from the applications modified. We attempt to illuminate our discussion by referring to brief code snippets from SSHD and Icecast listed in Figures 7 and 8, respectively.

1) Session Preparation: The first task in enabling session migration support for any server application is to associate appropriate handlers for disconnection and attachment events. For

```
main() {....
  /* Check if it is a migrate resumption process */
  mfileid = is_migrate_continuation();
  /* If resuming, call continuation function to restore state */
  if (mfileid) {
    ssh_session = migrate_get_session(mfileid);
    SSH_restart();
  }....

  /* Check for existence of migrate libraries */
  if(migrate_avail()) {
    ssh_session = migrate_get_session(main_socket);
    /* Registering handler function */
    migrate_register_handler(ssh_session,SSH_hndlr);
    }....
}

/* In quiescent point in main application
   processing loop ... */
void server_loop() {....
    /* Save process context */
   if(sigsetjmp(migrate_bs, 1) ||
    /* Check for outstanding suspend requests */
       migrate_suspend_now()) {
       /* Generate continuation and
          suspend application process */
       SSH_gen_cont();
   }
   else
       /* just keep track of saved context */
       migrate_bs_valid = 1;

   /* Sleep in select() until there is something to do */
   wait_until_can_do_something( &readset, &writeset,
                                ..., max_time_millisec);

   /* If starting some processing, reset saved context */
   migrate_bs_valid = 0;
   ...
   /* loop */
```

SSHD, at the onset of each application process (recall that each client is handled by a separate process), we associate a handler `SSH_hndlr()` for the main connection as shown in lines 13–14 of Figure 7. Icecast, on the other hand, is a multi-session application, so a handler is registered for each client session as it is established. Additionally, the Icecast also registers a global handler at server startup (lines 3-4 in Figure 8) to accept any clients that may elect to migrate existing sessions.

The next step in instrumenting a server application is identifying a quiescent point where the continuation can be generated and read. It was simple to identify quiescent points for SSHD and Icecast where the suspension and resumption could occur. In particular, after completing all pending tasks, both applications use the `select()` system call to block awaiting additional input, so generating a continuation at this point is straightforward.

For instance, the top of the `server_loop()` function was selected as the quiescent point for SSHD. As shown in lines 21–23 of Figure 7, the session checks for a suspension request once per iteration. (The process context is stored using the `sigsetjmp()` system call to allow immediate return to the quiescent point in case the application is blocked waiting for network activity.) The Icecast implementation is similar, checking for suspension requests each time it polls the media source for more data, as shown in lines 11–13 of Figure 8. It is important to note that the choice of quiescent point affects the latency of the resumption as that decides the frequency of the lookup after migration of session serving the session can not be resumed unless the point is reached. The ease with which the programmer was able to do this for these two applications, together with the prevalence of select-loop design for network applications, suggests that Migrate's session continuation approach may be a natural way to enable migration in many network applications.

Once a quiescent point has been instrumented, the handler itself is straightforward. A light-weight signal handler, it simply records the notification from Migrate, performs any necessary pre-processing, and allows session processing to continue unabated until the quiescent point is reached. In the case of SSHD, the handler may need to interrupt the blocking `select()` system call through `siglongjmp()`, as shown in line 47 in Figure 7.

2) Continuation Generation & Resumption: In contrast to the handler registration, the continuation generation functions are quite application-specific.

SSHD: Since SSHD handles each client session by spawning a new process, it requires C_{comp} to be generated at the suspension. The environment information necessary for process resumption is generated my `migrate_return_cont`. The information stored as part of the continuation include command line options, environment variables, and current working directory. We assume that system infrastructure of different potential servers to migrate are similar so that the above information is sufficient to resume the application at a different host.

In terms of preserving the application state, the continuation contains information about the main SSH session as well as any forwarded connections, such as X11 or TCP/IP ports. For listening sockets associated with port forwarding, the port numbers are stored in the Migrate-provided attribute/value store so that appropriate listening sockets can be opened at resumption. For all connections currently being forwarded, we associate dependencies between the main SSH session and the sessions containing them. As the number of forwarded connections grows, the associated state to be stored can grow relatively large. Additionally, to resume SSHD service across different hosts, the application state needs to include addition information, such as user login, authentication mode etc. Finally, Migrate generates appropriate resource continuation and makes them available before the application is resumed.

At resumption, Migrate starts new SSHD application using the continuation. When the application starts up, it checks the `MIGRATE` environment variable to decide whether the process is a resumed service. (Migrate adds this variable, along with several others containing configuration parameters, in the environment of every complete continuation.) If so, the process retrieves all the necessary session state stored in the attribute/value store and populates the appropriate data structures. While the resource continuation will ensure any forwarded connections are restored, SSHD itself must open a new pseudo-terminal and associate the terminal with the main ssh_session. Future versions of Migrate may provide built-in support for pseudo-terminal resource continuations.

Icecast: As mentioned earlier, Icecast is a threaded process with each client is associated with a thread. We generate C_{int} for Icecast sessions since suspension/resumption of any icecast session does not require shutting down and starting the Icecast process. We assume that Icecast service is available at all potential hosts for migration so that C_{int} is sufficient to resume service across hosts, otherwise appropriate C_{comp} needs to be generated.

For each icecast session, the C_{uint} needs to store the request identifier to look up the appro-

```
void source_func() { ....
    /* If migrate available, register attach handler */
    if (migrate_avail())
        migrate_register_global_handler(...)
    ...
    /* Process all clients for current source */
    while() {
        /* Attach waiting clients migrating here */
        if(migrate_attach > 0) client_attach();
        ...
        for(all current clients) { ...
            /* Suspend client with outstanding
               migration requests */
            if(client_to_be_migrated)
                icecast_gen_cont(session, socket);
            /* Proceed with regular processing */
        ...
        }

    }
}

void client_attach() {
  /* Restore client-related info. from migrate */
  migrate_client_restore(cli_session);
  /* Find source for the client */
  source = find_mount_with_req(request);
  /* Create client connection & associate with source */
  client = create_client();
  pool_add(client);
  client->source = source;
}

while() {
 if(migrate_attach > 0) client_attach();
  for(all current clients) { ...
    if(client_to_be_migrated)
      icecast_gen_cont(session, socket);
  } ...
}
```

priate source at the new server and the client's current offset in the playback. In addition to storing the appropriate information, the continuation generation involves performing appropriate bookkeeping operations to remove the client session from the active session for streaming and add the session to the list at the resumption time. The system continuation for icecast only includes the network socket used to server the client session and the data in the TCP buffer.

At resumption, whenever a client session is migrated, the global handler is invoked with the network socket of the migrated client session as an input. A new `client_connection` is created in response as shown in lines 23–30 of Figure 8. Next, the client's original request is used to look up the appropriate source for its streaming request, and the client connection is added to the list of connections for the server to stream to in future.

C. Migrate Daemon

The Migrate daemon runs with super-user privileges and is responsible for managing all open sessions on an end host; it registers each connection associated with a Migrate session with the available connectivity monitors and receives any changes in network availability or session connectivity. Upon receiving an event, the daemon consults the policy engine to decide how to handle it. Options include informing the application (the common case), forcing a migration (*e.g.*, when a cheaper network attachment point becomes available), suspending the session, or even delaying or ignoring the notification in highly variable conditions.

When managing a session, if either the owning application or the system policy engine informs the daemon a session should react to an event, the daemon contacts the remote end point. It does so by communicating over a separate TCP control channel that is created during session establishment or when resuming sessions. Control channel establishment is initiated by the connecting/migrating end point by sending a message to a well-known port on the remote host. [1]

1) Continuation Management: When an application generates a continuation, either by passing it as a return value from its mobility handler function or through the `return_cont()` function,

[1]Our implementation assumes that contacting different ports at the same IP address results in connections with the same end point. While this assumption generally holds in the direction of connection establishment, it may be not be true with some esoteric NATs.

the Migrate library considers whether the continuation should be executed immediately or passed to the Migrate daemon, where it may be composed with additional resource continuations.

To invoke an internal continuation, Migrate must temporarily interrupt the execution of the application process that generated the continuation and cause it to pass control to the function specified by the continuation. The Migrate daemon interrupts the application process using an asynchronous POSIX signal; the continuation function is invoked by a signal handler in the session layer library running inside the application process.

The Migrate signal handler sets the process signal mask to ensure that further mobility events will be queued until the continuation function has returned. It is further expected that continuations are not recursive (*i.e.*, they don't call `return_cont()` internally). These two conditions combined ensure that both mobility handlers and continuations are not re-entrant—freeing programmers from the need for synchronization or mutual exclusion.

2) Complete Continuations: The implementation of complete continuations is considerably more complicated than that of internal continuations. In the case of a complete continuation, Migrate terminates the original application process after the continuation is generated and instantiates a new one once session connectivity is restored. In many ways, the challenges faced are similar to those presented in process migration, except that the new process executes on the same host environment as the previous process. In both cases, the system must do one of three things for each resource used by the application process: transfer state from the original process to the new one, arrange for forwarding, or use similar state from the new process and sacrifice transparency.

Continuations are designed to limit the scope of the state and resources that must be transferred from the old process to the new and are explicitly *not* transparent. Only critical session state (indicated by its presence in the session's attribute/value store) is transferred to the new process. Similarly, only network connections and those file descriptors included in the continuation are forwarded to the new process. All other state from the original process, including code and data segments, open files, message channels, execution state, and the process control block, is discarded. In the interest of system security, however, some aspects of the process control block, such as the real and effective user ID, priority, and current working directories are transferred to the new process.

The working directory of the new process is set to that specified in the `cwd` field of the

continuation. Immediately before calling `execve()` with the `argv` and `envp` values from the continuation (`argv[0]` is treated as the executable), the process restores the signal mask in place in the original process when the continuation was generated. To assist applications in discovering the identity of the session being continued, the Migrate daemon adds two variables to the process' environment: `MIGRATE_CONT` contains a file descriptor corresponding to a resumed connection that is a member of the restored session; applications will likely use this value as the parameter to `get_session()` to recover the session itself. The second variable, `MIGRATE_PID`, contains the process ID of the previous process. Applications may wish to use the ID to communicate with orphaned children of the previous process, which presumably assumed control of the process group of the previous process.

3) Resource Continuations: When file descriptors are passed into the Migrate daemon as part of a continuation, Migrate considers generating resource-specific continuations to more efficiently preserve them. The daemon determines what resources individual file descriptors correspond to through the use of `getsockname()` and `fstat()`. Once aware of the resources in question, the daemon can generate resource-specific continuations. One trivial continuation is to close any redundant file descriptors (*i.e.*, multiple file descriptors to the same resource). Many interactive applications typically have three different file descriptors (`stdin`, `stdout`, `stderr`) that correspond to the same resource (generally a tty). Similarly, depending on the locking semantics in use (record, file, etc.), multiple file descriptors pointing to the same file can often be collapsed into one. Closing the only file descriptor pointing to a particular resource can be dangerous, depending on the semantics expected by the application. For example, closing all references to a file may allow the file to be subsequently deleted or overwritten. In the case of NFS, however, these operations can occur anyway, so Migrate could release all references without affecting NFS semantics. In order to ensure the operating system semantics are never affected, however, the current version of Migrate always keeps at least one file descriptor open for each (non-network) resource contained in a continuation.

D. Session Control Protocol

As presented before, session migration generally consists of four parts: end point location, authentication, rebinding (including any required port mapping), and connection synchronization. Additionally, if the client elects to migrate to a new server, a fifth, validation stage is required to

ensure the replica server is capable of servicing the existing session. Migrate's session control protocol supports each of these five operations.

1) End Point Location: The main task of the protocol is to manage changes in network attachment point. When either end point changes attachment point, Migrate must update the remote end point of each open session with its new network attachment point. When an end point moves, the selection of a new remote attachment point is governed by the lookup function provided to Migrate as described above.

2) Authentication: The main difficulty with updating the remote end points is validating the end points—ensuring that the end point requesting the update is the original end point. Migrate relies on the application to authenticate the end points before session establishment. Once authenticated, Migrate negotiates a shared symmetric key during session establishment using a Diffie-Hellman key exchange [4]. The application-authenticated end point uses this shared key to authenticate session updates after a migration event. We assume that all candidate replica servers available to the client are both connected and trusted. Hence, in the event a replica server receives a migration request from a client it was not previously serving, it can obtain the session state including the keying material directly from the original server itself.

The details of the remaining stages of the session control protocol, including connection rebinding and synchronization, vary depending upon the connection migration technology in use, and are described in detail elsewhere [22], [23].

When resuming a session at the same host, the authentication process takes about four round-trip times (RTTs): a TCP SYN/SYN-ACK exchange; a session resumption request; a challenge and the corresponding response from the end point seeking resumption; and bidirectional messages to map any suspended connections (ports) to new ones. For migration of the session to a different host, communication between the Migrate daemons between the old and new hosts is needed to ensure synchronization and transfer of state information while resuming a suspended session at a new host. The protocol consists of messages exchanged by the Migrate daemons on all the hosts involved, as shown in the example in Figure 9. The figure illustrates the control messages exchanged when client moves a single Migrate-enabled connection from the old server to the new server.

Migration begins when the client queries a potential new server with a `SESSION_TRANSFER_REQUEST` message which starts the first stage, *client authentication*. The message carries the server address

and the session id from the original session, which are used by the new server to look up the old server and obtain the session authentication key over a secondary control channel. The key is used by the new server to authenticate the client, upon success of which the new server sends `COMPATIBILITY_CHECK_REQUEST` to the old server and proceeds to the next stage, *Server Dependency Verification*.

After receiving the request, the old server freezes the original session and any dependent sessions if they were still running, as a results generating the continuations and backing up the attribute/value store. The dependency vector is extracted from the store, and transmitted to the new server for the verification against the registry of compatibility parameters there, as explained in Section V-A. The suspension of the session is necessary to to guarantee consistency of the compatibility information with the current state of the application. The client is notified of the result of the verification via the `COMPAT_SUCCESS/FAILURE` message, allowing it to decide, in accordance to the local migration policy, whether to continue migration of the service to this server. The daemon at the old server uses state information associated with the session to ensure that only a single new server can proceed to session migration.

The next stage is initiated by the client sending a `SERVICE_MIGRATE_REQUEST` to the new server. The new server, in response, starts transferring the state information for the main session and any dependent sessions from the old session and the session at the old server is closed. The state information consists of the application's attribute/value storage file, the file containing the continuations, as well as the state of the network connections saved by Migrate. To reduce the state transfer latency, these files are compressed during transfer. After transfer, the files are updated with server specific information, and a `CLEAR_TO_MIGRATE` message is sent to the client. The client can then resume the session anytime by sending a `SESSION_RESUME_REQUEST`, and in response the new server unfreezes the session and executes the continuations. The Migrate daemon on the new server then resumes the session locally, also translating the id of the of the migrated session (and updating the session-layer library) in order to avoid collisions with existing sessions on this host. It then updates the changes in the session information to the client with a `SESSION_UPDATE` message, which completes the migration process.

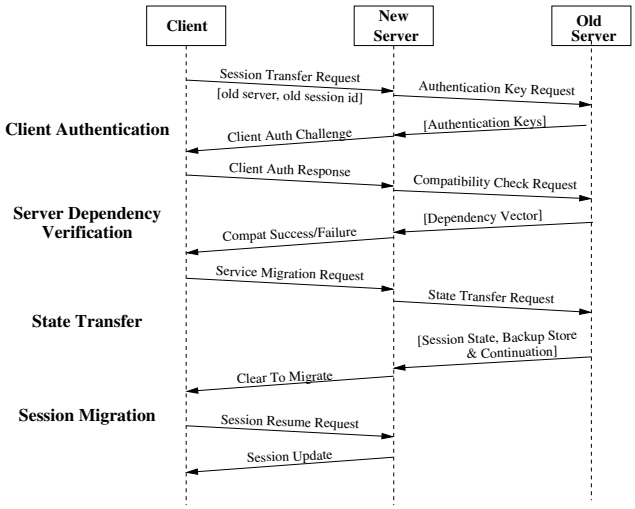


Fig. 9. Stages in Session Migrate Protocol

```
# A sample Migrate policy file

monitor-policy {
    score-interface * 1
    score-interface eth* 10
    score-interface eth0:1 1000
    score-interface ppp* 5

    on proto tcp {
        on remote-ip 18.31.0.4/24
            on port ssh
                score-interface eth* 100
        on l-port 3001-3010 {
            score-interface eth* 200 ;
            score-interface eth0:1 100
        }
        on port http, ftp
            migrate never
    }
}
```

Fig. 10. A sample Migrate policy file.

E. Policy Engine

Ideally, Migrate-aware applications would assist the user in managing mobility preferences and coordinate them with Migrate through a policy API. Such an interface could allow for dynamic adjustments to local policy based on the particular activity the application was engaged in at any particular moment (*i.e.*, support the suspension of important users' sessions for longer periods of time, etc.).

Unfortunately, most applications currently lack direct interfaces to allow users to describe mobility handling preferences. Therefore, Migrate provides for an optional rule-based user policy file in which users can express their preferences in terms of which local network attachment point to use for particular applications. Migrate currently requires users to specify sessions based on the transport-layer ports. For example, if a user wished to specify policy for Web browsing, she would insert a policy rule for the TCP protocol on the well-known HTTP server port (80).

Migrate consults the policy file at every mobility event: the creation of a session, a change in the set of available local attachment points, or change in the connectivity status of a particular session end point (tracked as described in the following section). In each case, the policy file answers the question, "Which local attachment point, if any, is best suited for a particular session?" Each session is considered in turn. The input for each invocation is a description of the network connections comprising the session in question—their protocols and current remote and local attachment points, including ports. The file directs Migrate to take one of two actions: either migrate the session to a new interface, or leave it alone. For each available attachment point, a score is computed; the attachment point with the highest score for a particular session is selected, provided the score is sufficiently higher than that awarded to the current attachment point.

Figure 10 shows a sample Migrate policy file. In this example, `eth0` is preferred to other `eth` interfaces, which are preferred to `ppp` interfaces, which are preferred to all others. Sessions containing TCP SSH connections to remote attachment points with IP addresses in the 18.31.0/24 subnet have an increased affinity for `eth` interfaces, and TCP connections on local ports 3001–3010 actually prefer `eth0` less than other `eth` interfaces. Finally, sessions containing TCP connections to HTTP or FTP server ports are never migrated.

Connectivity monitors can function at various levels of the protocol stack. For example,

Migrate's physical layer monitors glean connectivity information from network interfaces (*i.e.*, whether the cable is plugged in, whether the physical device is operating properly, etc.). At the other extreme, applications themselves may gather connectivity information, perhaps by passively monitoring their connectivity, or through active probing (*e.g.*, keep-alive probes).

The network and transport layers can provide especially useful connectivity information. For example, Migrate provides a connectivity monitor that uses TCP retransmission timer status to assess the connectivity of sessions containing TCP connections. If an outstanding byte is not acknowledged, TCP will attempt to retransmit it after a period of time known as the retransmission timeout (RTO).[2] The lack of a response to subsequent retransmissions leads to an exponential increase in the RTO. Hence, the "health" of a TCP connection can be described by the value of a connection's RTO—the larger the value, the greater the connection's distress.

VI. Evaluation

In this section, we present results of our experimental evaluation of the proposed system for suspending and resuming network applications. We evaluate continuation-based application migration along four dimensions. First, we discuss the complexity of the code changes necessary to the applications to enable continuation-based session migration. Then, we study the performance of session continuation as the building block to enable efficient application migration. We next discuss the overheads associated in enabling the migration functionality. Finally, we evaluate scalability of the proposed session migration techniques.

In our experiments, we evaluate a C/C++-based Migrate implementation running on various versions of Linux. We evaluate our proposed system by extending two very popular session-based Internet server applications, OpenSSH 3.0.2p1 [26] and the Icecast streaming media server v1.3.10 [8]. In addition, we use a simple client-server benchmark program for some of the experiments to evaluate performance without any application-related overhead. For our experimental test-bed, we used two machines with similar configuration (2.8-GHz Pentium4 with 256 MB of memory) connected by a 100 Mbps Ethernet link for our performance testing. The measured round-trip latency between the two servers is 0.17 ms. We timed Migrate using its

[2]In fact, a TCP sender may attempt an earlier retransmission in response to a duplicate ACK. The RTO is a persistent, timer-based retry mechanism used as a fail-safe.

built-in TCP virtualization support; the TCP migration options were not supported by either kernel.

A. Code Complexity

Application	Original LoC	Same-host LoC	Overhead (%)	Separate-host LoC	Total Migrate LoC	Overhead (%)
OpenSSH	38,885	459	1.18	201	660	1.69
Icecast	18,676	43	0.23	60	103	0.51

TABLE II

CODE MODIFICATIONS REQUIRED TO ENABLE APPLICATION MIGRATION USING THE MIGRATE TOOLKIT.

While the brief code snippets in figures 7 and 8 can provide some visibility into the modifications required, they cannot accurately convey the scope or complexity of the complete set of modifications required. Indeed, quantitative evaluation of coding complexity is quite problematic. For lack of other better measure, we present the lines of code (LoC) added in order to enable the migrate functionality.

Table II presents the lines of code added for SSHD and Icecast applications for same-host resumption as well as separate-host resumption. For comparison, we also report the total LoC for each application. It is observed that for same-host migration, the SSHD application required an overhead of 459 lines of code, while Icecast requires only 43 lines of change. While the difference is partly due to the difference in the complexity of the applications, it is also due to the fact that SSHD generates a complete continuation while Icecast generates an internal continuation. Also out of 459 lines of code for the SSH-migration, 226 lines are for ensuring all dependent channels (such as X11, TCP/IP, and agent forwarding) are resumed/stored accordingly, which is not required if only one ssh-session needs to be migrated. Separate-host migration needs additional code modifications to generate appropriate continuation for local resources and steps to reallocate resources at the destination server. In this case, the additional overhead is proportionately much less for SSHD (201/459) than Icecast (60/43) since less additional state needs to saved/transferred.

In general, the majority of the code modification is for marshalling and unmarshalling the session state through the Migrate attribute/value store. Only 80 lines of code deal with the

control flow related to service migration. The marshaling code is highly stylized and could be considerably optimized through judicious use of macros to expunge repetitive code. We have not, however, made any attempt to optimize code size. Instead, we present measurements of the modifications as they were first implemented in an attempt to provide a more accurate view of programmer effort. In contrast, the level of code modification required for Icecast is considerably less, adding only 103 lines of code of which 60 lines are devoted to operations required to resume the thread at the destination server.

While the lines of code represents one part of the code complexity, the time for code modification depends on the familiarity and expertise of the code base for the service application. In our experience, it was much easier to introduce the migrate functionality to Icecast server as compared to the SSHD server because of the low complexity of the server, threaded design, and clean abstraction of code for easier instrumentation. For reference, we note that the size of our code modifications compare favorably with existing mobility toolkits such as service continuations [25], which required more than three times the number of changes to Icecast (350 LoC), and Rover [10], which reported requiring changes to between 10 and 15% of an application's code base.

B. Migration Performance

Having discussed the code modifications required for enabling continuation based migration, in this subsection, we evaluate the performance of migration in terms of resource conservation and the latency of migration.

1) Resource Conservation: Session continuations allow hosts to conserve scarce resources during periods of disconnection. Conserving system memory and file descriptors is especially important for servers that are designed to handle a large number of concurrent sessions, because a significant number of the current sessions may in fact be suspended.

Figure 11 shows the memory footprints of processes serving active SSH and Icecast sessions. These values are obtained using gcc version 3.2 with the `-O2` option on a Linux 2.4.20 system with 1 GB of RAM and 1 GB of swap. For comparison, the processes are shown with our Migrate-extensions and without. The Tesla stub required for Migrate support increases the memory usage of both applications. The Icecast process has a smaller increase because it already loads many of the libraries required by the Tesla stub. During suspend, in case of SSH the process is closed

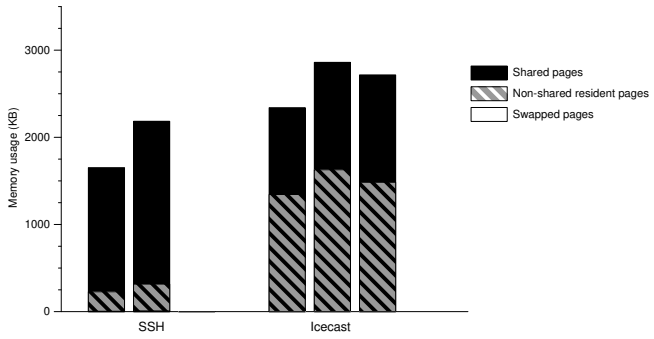


Fig. 11. The memory footprints of sample Migrate-aware servers. For each application, the first column represents the unmodified application and the middle corresponds to the Migrate-aware version, and the last for suspended state of application.

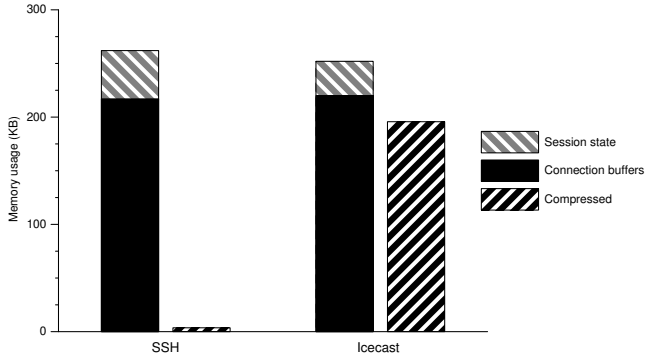


Fig. 12. Complete continuation sizes in KBytes. The sizes reported here include persistent application state, buffered network connection data, and all associated Migrate control data necessary to invoke the communication.

leading to a lot of memory conservation. On the other hand, the memory save for Icecast is less as the application is kept alive while conserving the memory consumption by the client session being suspended.

In both cases, the complete continuations generated upon disconnection are 3.6 KB and 195 KB, respectively. The difference in the size of the continuations is due to the much larger connection buffers required by the Icecast session, as shown in Figure 12. The Icecast continuation also includes the name and properties of the stream, the user identity information etc. The SSH continuation includes the session keys and any unsent data in the encryption buffers. For each application, an uncompressed continuation is shown on the left, the compressed version on the right. Moreover, because the suspended SSH session had just started up, its connection buffers were largely unused, resulting in much more effective compression. The Icecast session's connection buffers were filled with less-easily compressed file data (compressed audio files), resulting in a substantially larger continuation.

Table III shows the number of file descriptors required for an individual session in each

Name	Native	w/Migrate	Conns	Suspended	Comp.
OpenSSH	9	12	1	8	3
Icecast	17	20	2	2	16

TABLE III

THE FILE DESCRIPTOR USAGE OF SSH AND FTP SERVERS

application for same-host migration. The SSH session in this example is logged in to a command shell, while the Icecast session is in the middle of streaming audio to two clients. The first two columns indicate the number of file descriptors used by an active session before and after enabling Migrate support. The third column shows the number of these descriptors corresponding to active network connections. The last two columns present the number of file descriptors required for sessions suspended through a continuation. The "Suspended" column indicates the number of descriptors included within the continuation, and the "Compressed" column shows the actual number held open by Migrate during disconnection after generating all available resource continuations.

Migrate generates simple resource continuations for these applications by closing redundant descriptors and leaving only one descriptor pointing to a particular resource. This reduces the number of descriptors used in the suspended state to three and sixteen, respectively, as shown in the last column of the Table III. The three descriptors for SSH correspond to the connection to the user's shell, its pseudo-tty, and the `/dev/ptmx` device.[3] In the case of Icecast, most of the file descriptors remain open since the process remains active serving other clients or waiting for new clients to attach. The active file descriptors of Icecast include connections to all media sources, files opened for collecting statistics, a console terminal, and connections to server other existing clients. The file descriptors related to the client session being suspended is included in the continuation. The savings in file descriptors can be very high when the number of suspended client sessions is large. Note that, in case of separate-host migration, all the non-network file descriptors are closed and appropriate continuation is generated to resume them at the destination server.

[3] `/dev/ptmx` is a device used to control the allocation of pseudo-ttys on Linux.

STAGE	Migration Latency (ms)							
	Same host				Different host			
	benchmark	SSHD	SSHD w/fwd	Icecast	benchmark	SSHD	SSHD w/fwd	Icecast
Appl. Suspend	0.27	0.35	0.47	61-99	0.25	0.35	0.51	61-99
Serv. Authentication					0.88	0.81	0.86	0.80
Dependency Check						0.21		0.47
State Transfer					9.93	10.88	18.94	54.01
Cli. Authentication	0.85	0.75	0.75	0.62	0.48	0.48	0.48	0.49
Migration	0.52	0.34	0.34	0.39	0.48	0.66	0.66	0.60
Appl. Notification	54.23	56.66	114.92	2.31	56.23	56.25	112.73	67.11
Appl. Service Delay	0.02	1.38	1.40	88-170	0.01	1.38	1.38	13-206
Appl. Continuation (Appl. Resume)	1.21	1.23	1.16	0.86	1.01	1.38	1.53	2.93
End-to-End Latency (Suspend+Resume)	57	61	119-274	153	69	72	137	200-331

TABLE IV

SESSION MIGRATION MICRO-BENCHMARKS FOR SSHD, SSHD WITH FORWARDED TCP CONNECTION, AND ICECAST. THE VARIATION IN THE LATENCY FOR ICECAST IS DUE TO APPLICATION DATA BUFFER SIZE AND VARYING APPLICATION STARTUP TIME FOR ICECAST.

2) Migration Latency: In this subsection, we present experimental evaluation of migration latency for three applications, i.e. simple client-server benchmark, SSHD, and Icecast. In addition to using two well known network applications, we implemented a simple client-server benchmark application to minimize application-related complexities. In this application, the server sends periodic updates with index numbers and the client prints the index of the update. In this example, at the suspension time the server needs to remember only the index of the update so that it can resume from the point it left. Note that the suspension of the server generates a complete continuation.

Table IV presents micro-benchmarks of the time spent in different stages of the migration operation when a server session of each application is resumed at the same-host as well as migrated to a separate host. The reported values are based on five runs for each test case. In our tests, the delay experienced during service migration is quite small leading to a seamless user experience. The client-server benchmark takes 57ms and 69ms for same-host and separate-host

migration respectively. The full SSHD migration takes 61 ms where as migration of SSHD with one TCP forwarding takes 119ms. The standard deviation of the measured latency for SSHD is within 3% of the mean. However, we observed a high variation across multiple runs in the migration latency for Icecast. For better understanding, we report the range of time spent for the stages that have a standard deviation of more than 5% of the mean; otherwise, report the average time. We will explain the reason for this variation later in this subsection.

Table IV reports latency of the different stages in the suspension and resumption of a server process, starting from suspending process till resuming operation after migration. The first step shown is the *Application Suspend*, which takes a small time for the single simple client-server application as well as SSH process, increasing marginally when a dependent session for the forwarded connection needs to be suspended. However, the time needed for Icecast is large, because here the suspension is for one thread out of several which are processed together. The delay observed depends on how long after the suspend is triggered the application gets around to handling that thread, which depends on the initial state of the application when the migration is triggered.

The remaining stages are for migration and resumption, out of which the initial stages apply only for migration to a separate host. The first stage, *Server Authentication*, consists of the initial message exchange to request migration and the necessary authentication step. Next, the compatibility check stage, *Dependency Check*, validates the feasibility of resuming the service at the destination location; hence, the time spent in this stage changes depending on the particular dependency requirements of the service under test. The current implementation has basic compatibility checks, but this is likely to change depending on the service requirements.

The next stage, *State Transfer*, refers to the transfer of the session continuation and state files across the network from the original server to the new server. The duration of this stage strongly depends on the size of each of these state files as well as the round-trip delay between the servers. The size of the attribute/value store varies depending on the minimum amount of application state information to be stored so that the service can be resumed appropriately. Of our applications, SSHD needs to save substantially more application state information than Icecast. For instance, the compressed size (Migrate automatically compresses the attribute/value store using gzip) of SSHD's store is 3.9 KB where as Icecast's is only 1.6 KB. Similarly, the size of the resource continuation depends on the amount information that needs to be preserved per

network connection including the TCP buffers, the state of the connections. Icecast tries to buffer a large amount of data for future playback in the TCP buffers, hence takes a long time to transfer the resource continuation (94 ms). The situation can be improved further by ignoring storage of buffers while suspending the connection, especially if the new server is going to move the client directly to the live portion of the feed or has historical feeds available directly.

The steps following the state transfer are common to both same-host and separate-host migration. The first amongst these is the *Client Authentication*, during which the migrate daemon on the host initiating the migration connects to the daemon on the host where the continuation is to be executed, and authenticates itself. Authentication is followed by *Migration*, which corresponds to the local Migrate daemon's internal bookkeeping involving multiple sessions' state transfer from the previous Migrate daemon. Hence, depending on the number of sessions to be transferred (depending on the session dependencies) the latency would change for this stage.

The initial phase of session continuation, *Application Notification*, involves resuming application's session related information, synchronization with the local migrate daemon, and restarting the application process in case of complete continuation. It can be observed from the table that application resumption time is less for Icecast (2.31ms) as compared to the SSHD (56.66) because in the case of SSHD the server process is restarted at the destination whereas Icecast need only contact a thread upon migration. Moreover, the time spent in this stage also depends on the complexity of the service such as number of sessions, resource continuation issues like the size of the TCP buffers sizes dumped before suspension, and any session dependencies. For instance, the *Application Notification* for SSHD with TCP/IP forwarding takes longer (114.9ms) than base SSHD resumption, since it involves resumptions of two dependent sessions. The time needed for resumption at a different host is almost same for SSH which is resumed from complete continuations. However, the time required for resuming Icecast because of modifications that need to be made in the Tesla data structures when the migrated session is attached to an existing process.

The next phase (*Application Service Delay*) is the time between application is notified (or resumed) till it reaches the quiescent point to start the application's continuation. It strongly depends on the structure of the program and the decision of quiescent point. It can be noticed that the Icecast has a large variation in the latency in this stage. The primary reason is the structure of source loop in Icecast and the choice of quiescent point. In Icecast, as we decide

to check for new clients at the beginning of source service loop, where the delay between two checks is a function of current number of clients to service, and latency of one chunk of data read from the source server. Additionally, due to a bug in the pthread implementation in Red Hat Linux, individual threads cannot be signaled. Hence in our implementation, we receive asynchronous events in one thread, and set a flag for the source thread to check later which may add overhead and variation in the time *Application Service Delay*. The last phase, *Application Continuation*, executes the session continuation.

C. Performance Overheads

We study the impact Migrate has on TCP connection establishment latency, measure Migrate's session migration latency, and measure the bulk transfer throughput of Migrate-enabled TCP connections.

1) Computation overhead: Migrate increases the processing latency for a TCP connection by a small amount. We measured the processing latency as the elapsed time between the issuance of a blocking `connect()` system call and its return for a connection on the loopback interface. We give the results for three different types of connections on a 600 MHz Intel P3 running Linux 2.4.1: a native TCP connection, the first virtualized TCP connection of a Migrate session, and a subsequent virtualized TCP connection on a Migrate-enabled session. The median latencies observed over 100 independent runs were 0.16 ms, 3.3 ms, and 0.43 ms, respectively. As expected, the initial connection establishment on a Migrate session is significantly slower than subsequent connections, which are marginally slower than native TCP connections due to the Tesla IPC overhead. Cryptographic operations account for the vast majority of the session establishment overhead, requiring almost 3 ms in this configuration. The sessions in this experiment were secured using a 128-bit key negotiated using Diffie-Hellman, whose latency increases with key length. In practice, however, for wide-area Internet connections the connection establishment latency will continue to be dominated by round-trip propagation time, which is large compared to the additional latency introduced by Migrate.

Session migration consists of four parts: end-point location, authentication, rebinding (including connection port mapping), and connection synchronization. Because the delay associated with end-point location depends on the naming system selected by the application, we assume here that the location of the remote end point is known—which is the case unless both end points

move simultaneously. The cost of connection synchronization depends on the transport protocol in use and the loss rate experienced at the previous attachment point. The synchronization performance of both the TCP Migrate options and Migrate's TCP virtualization mechanism have been reported previously [23]. Here, we quantify the expense of the other two operations—authentication and rebinding—by measuring the migration latencies of sessions containing a variable number of connections. We define *session migration latency*, measured at the migrating end point, as the time between the attempted reestablishment the session control connection and the actual reestablishment of all of the associated network connections.

We first examine session migration latencies over a loopback interface to ignore the effects of network latency. We find that the latency to migrate n connections on an 850-MHz Intel P3 running Migrate on Linux 2.4.2 is $(1.6 + 0.55n)$ ms. End-point authentication takes about 1.6 ms, independent of the number of connections, and migrating each connection adds about 0.55 ms of delay. The per-connection delay variation increases with the number of connections because of the increased likelihood of context switching at one or both of the end points between the independently-sent connection mapping messages.

We also measured the median time required to migrate a session containing one virtualized TCP connection to an attachment point with a varying RTT between itself and the remote end point. Because session migration is an end-to-end operation, the delay depends mainly on the RTT between the attachment points used by the two end points. The measured latency is roughly equal to four RTTs. For n concurrent connections, the resuming host initiates resumption negotiations in parallel, so the overall session migration latency would be $(4 \text{ RTT} + 1.6 + 0.55n)$ ms.

2) Networking overhead: Migrate uses the TCP Migrate options if they are available in the communicating stacks, in which case there is no throughput degradation compared to standard TCP. To evaluate the impact of Migrate's connection virtualization on network communication, we measure the throughput of a virtualized TCP connection that is part of a Migrate session using the popular `ttcp` utility. Figure 13 shows the connection throughput for a native TCP connection and for a Migrate-enabled connection as a function of the block size of the `write` call done at the sender. This connection runs over a 11 Mbps shared 802.11b wireless LAN. Migrate introduces a noticeable overhead at small block sizes. The overhead is due to several factors including the additional context switch and system call overhead imposed by Tesla and the memory copy operations required by Migrate to virtualize TCP connections. Once the bottleneck

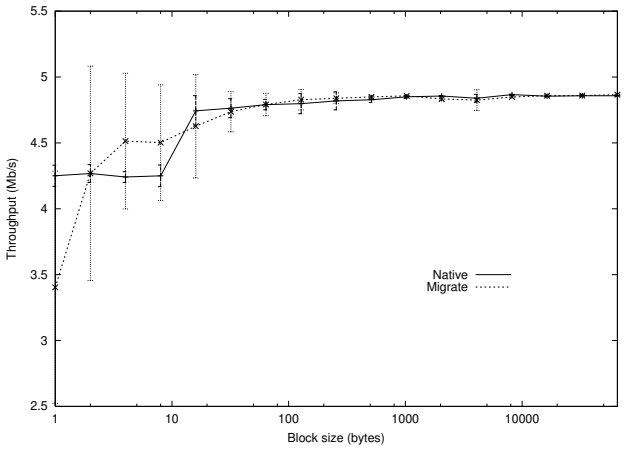


Fig. 13. Mean TCP throughput with and without Migrate on a shared 802.11b wireless LAN, as measured with `ttcp`. The receiver is an IBM ThinkPad T21 (600-MHz P3) running Linux 2.4.16 while the sender is an Intel 2.26-GHz P4 running Linux 2.4.18. Each point represents the average of at least sixteen runs; error bars represent one standard deviation.

becomes the available network bandwidth, the throughput reduction is less than 2%. For a similar throughput experiment over loopback we find that the throughput of a Migrate-enabled TCP using user-level virtualization is 350 Mbps on the measured platform.

Somewhat surprisingly, Migrate outperforms standard TCP connections for small block sizes. This is because of the interactions with Tesla; data flows through a Tesla process that runs in a loop that reads up to 8 KBytes at a time from the application independent of the actual application `write` size, and writes it to the network as one block, thereby not sending small packets on the network.

The larger throughput variability of Migrate-enabled transfers is due to the increased impact of context switching, cache replacement policies, and other scheduling vagaries on both sender and receiver. This scheduling effect impacts all Migrate operations, including data transfer, connection establishment, and session migration. Since all network I/O handled by Tesla must pass through a separate process, performance depends on how the application and corresponding process are interleaved. If this variance became a serious concern, it could be greatly reduced by invoking the Migrate handler as a co-routine or part of the system call itself.

D. Scalability

In this subsection we present results from our experiments to evaluate the scalability of the proposed system for resuming suspended sessions at the same host or at a separate host. We evaluate scalability for three applications: a simple client-server application, OpenSSH and

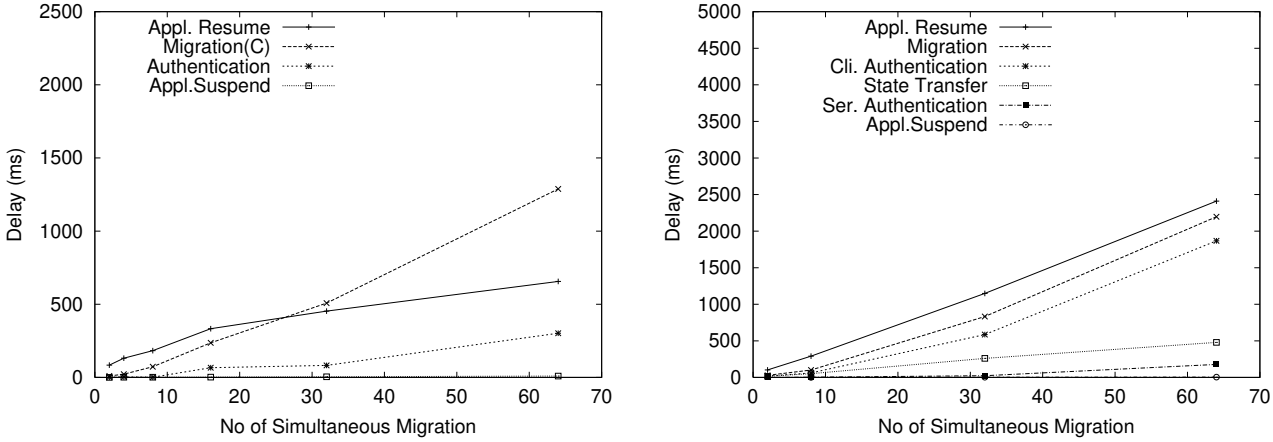
Icecast. In our experiments, we suspend and resume a number of sessions simultaneously, and observe the latency of suspension and resumption for each session. We use the average latency of suspension/resumption for sessions as the metric of scalability.

To understand the latency variation in depth, we also observe time consumed in different steps of migration. It is important to note that the time consumed in each step depends on the inherent computational/communication load of the step as well as the load on the Migrate daemon by other simultaneous migration requests at the moment. For resuming session at the same host, we report *Appl. Suspend* (application suspension time), *Authentication* (time consumed in authenticating the migration request), *Migration* (time consumed in the session to become established after migration) and *Appl. Resume* (when application is ready after complete migration). For resuming sessions at a separate host, we report the time consumed in two additional steps, i.e. *Ser. Authentication* (time spent in authenticating target server) and *State Transfer* (latency in transferring required state information to resume service at the new server). Each experiment is repeated 3 times, and we report latency averaged over three runs unless otherwise mentioned explicitly. We modified the code of Migrate daemon and the applications to compute the latency in each step of suspension/migration using `gettimeofday()` APIs.

1) Base case: Simple Client-Server Application: We use a simple client-server based micro-benchmark application to understand the overhead of the migration/continuation process when the amount of state information associated with the application is minimal. Figures 14(a) and 14(b) show the increase in suspension latency and resumption latency as the number of simultaneous suspend/migrate requests increase from 2 to 64. Figures 15(a)-15(d) present the variation observed in the latency across different sessions under observation. We will show variability for one test-case as the variability in latency experienced for other applications is similar to the sample client-server application.

First of all, it can be observed that the average application suspension time is very minimal (maximum of 9ms for 64 suspensions) and does not increase very rapidly as the number of simultaneous suspension requests increase. Figure 15(a) shows the variation observed in the application suspension stage. The variation in the suspension time is observed to be small.

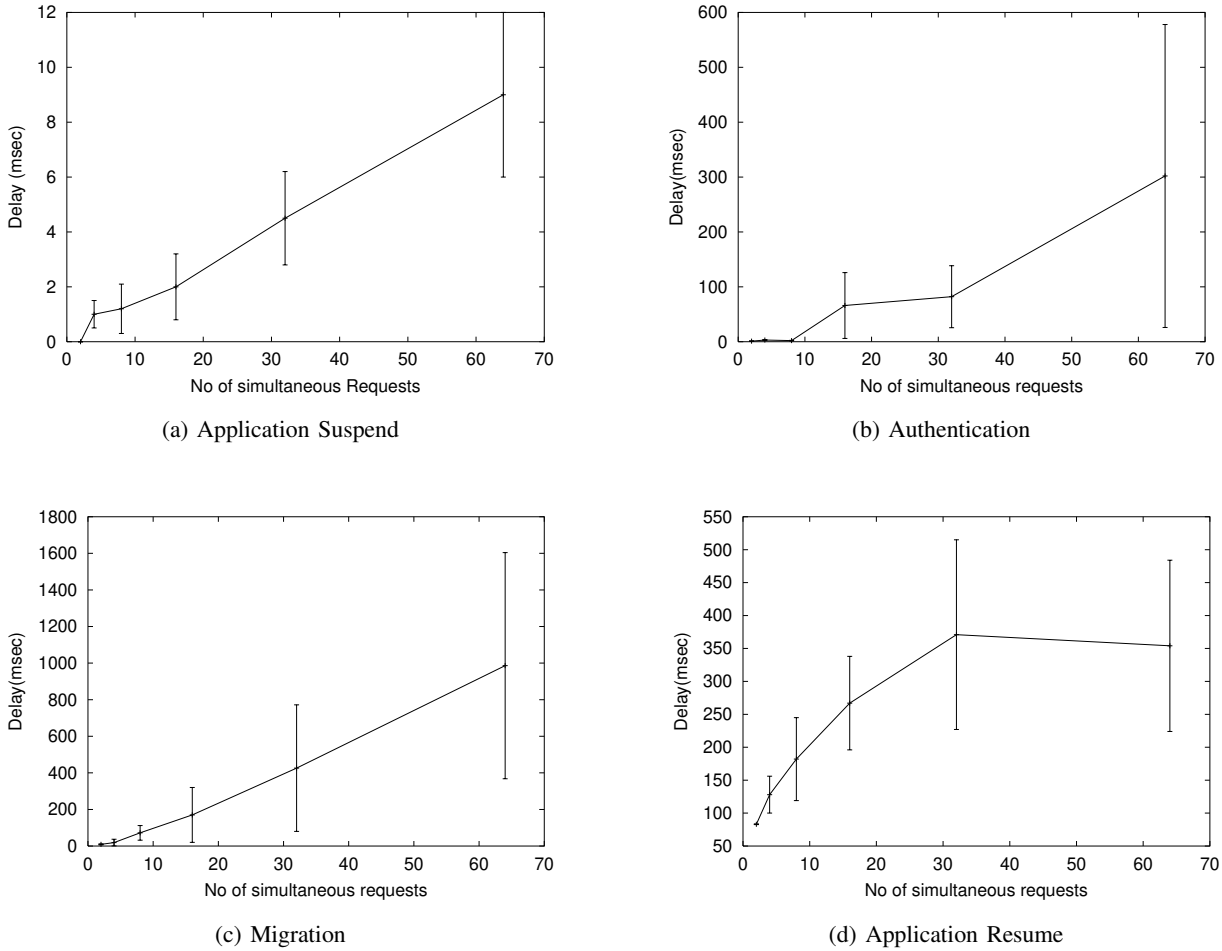
In resuming session at the same host, the client authentication is the first step. It can be seen from Figure 14(a) that the average time consumed in authentication step increases with number of simultaneous suspensions. This is due to the fact that as the number of requests increase the



(a) Resuming at same host

(b) Resuming session at a separate host

Fig. 14. Session Migration Latency for Simple Client-Server



(a) Application Suspend

(b) Authentication

(c) Migration

(d) Application Resume

Fig. 15. Latency Variation for Same Host Migration Simple Client-Server in Application Suspension, Authentication, Migration and Application Resumption. We report cumulative latency for each stage computed from the start of suspension/resumption. Each point represents average latency across all session and error bars represent one standard deviation

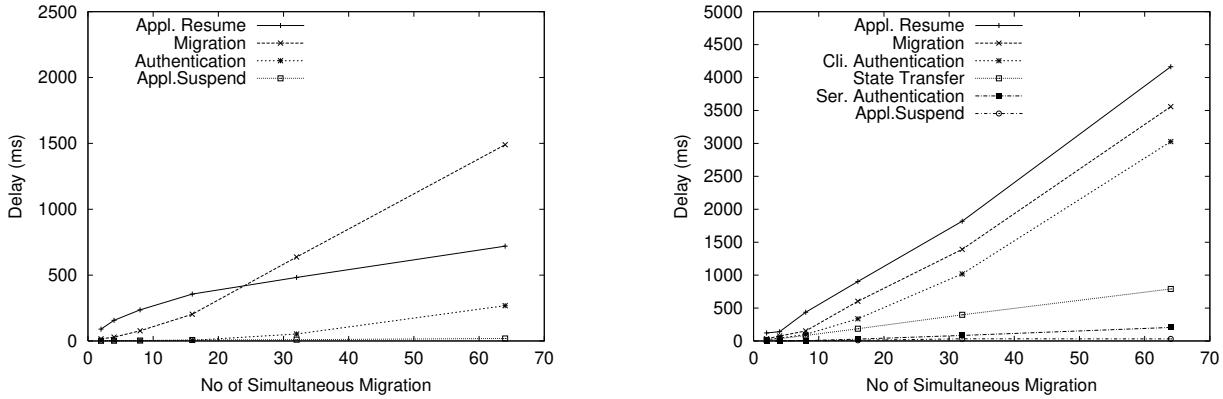
computational load on migrate daemon associated with the authentication (encryption) increases, affecting the response time for other incoming authentication requests. Additionally, it can be observed from Figure 15(b) that the latency of authentication increases rapidly as the number of simultaneous requests increase. This is due to the fact that as the initial requests are being processed with computationally heavy encryption task, the later requests have to wait before being processed.

After authentication step is over, two steps occur simultaneously, i.e. application resumption using the continuation information and the migration of sessions to the ESTABLISHED state. It is important to note that the application is ready to resume operation when both of the above steps is complete. It can be seen from the figure that the latency of application resumption is less than the migration for less number of simultaneous resumptions where the application resumption time dominates for higher number of resumptions affected by the inherent overhead of simultaneous process creation. Note that the total server latency increases linearly to the number of simultaneous session requests, hence does not show any inherent capacity/scalability bottleneck due to migrate.

Figures 15(c) and 15(d) present latency variation to complete the migration as well as application resumption. Please note that the variation observed in these stages are dominated by the variation observed in the previous stage, i.e. authentication. Additionally, the process creation also introduces additional variability for simultaneously resuming the application.

For separate-host migration, two other important steps to note are *State Transfer* and *Ser. Authentication*. As mentioned earlier, the time consumed in state transfer is going to be affected by the size of application continuation and migration state files. We use standard compression techniques to reduce the volume of the state transfer, however leads to increase in the load on migrate daemon. Hence there is a perceived increase in the time spent in *State Transfer* though the application continuation size is very small. The time consumed in this step can be further optimized by selectively deciding to use compression thereby reduce the load and the average delay experienced in this step.

We observe that the time consumed in the server authentication process is similar to the client authentication step; however is dependent on the round-trip-delay between the old-server and new-server which is minimal in our setup. Another observation is that the average time consumed in client authentication while doing a separate-host migration is very high compared



(a) Resuming at same host

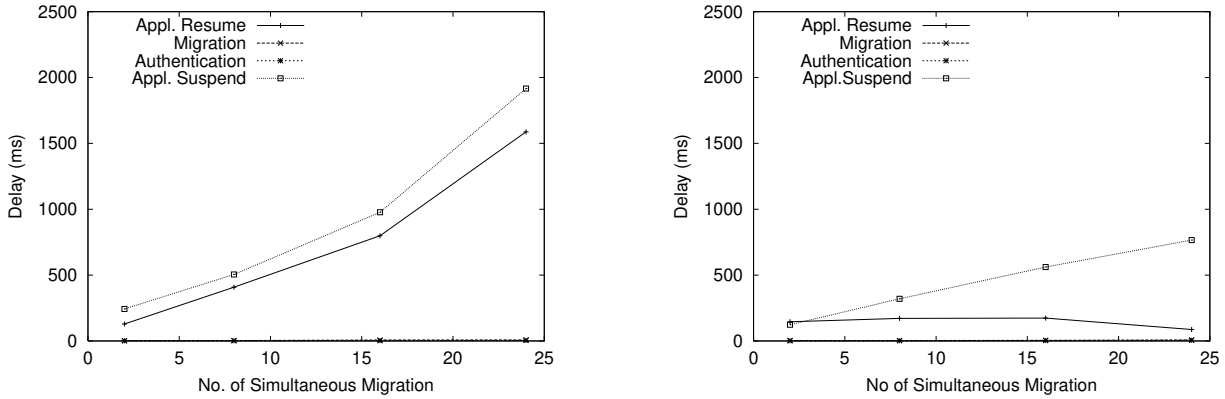
(b) Resuming at different host

Fig. 16. Session Migration Latency for OpenSSH

to doing it on the same host. This is due to the fact that at the time of client authentication the migrate daemon might be loaded with authentication tasks of other sessions delaying the response time for client authentication requests. Similar to the client continuation, it can be seen that the overall latency of resumption varies linearly with respect to the number of simultaneous session resumption requests.

2) OpenSSH: We conducted similar capacity analysis for OpenSSH application. Figures 16(a) and 16(b) present results of the experiments. The latency observed in same-host migration is very similar to the sample client-server application presented above except for the application resumption time. The application resumption time is slightly higher than that of simple client-server application. The application resumption time consists of starting new processes, reading from the continuation file and resuming the application using continuation. While creating new process would be similar to the simple client-server application, the other two steps potentially are more complex for SSHD decided by the amount of continuation information stored and continuation steps to resume. For migration of OpenSSH to a separate host, the trends observed in time spent in different steps are similar to the micro-benchmark as expected. In summary, it can be seen that we can support 64 simultaneous SSH continuation within 1.5sec and the service resumption latency varies linearly with respect to the number of servers.

3) Icecast: For evaluating Icecast continuation scalability, we set up two Icecast servers at two different servers and tried to suspend/resume multiple player sessions (using mplayer) at the same server as well as at a different server. We varied the number of simultaneous session suspension/resumption from 1 to 24. We could not perform experiments beyond 24 simultaneous



(a) Icecast

(b) Icecast without buffers

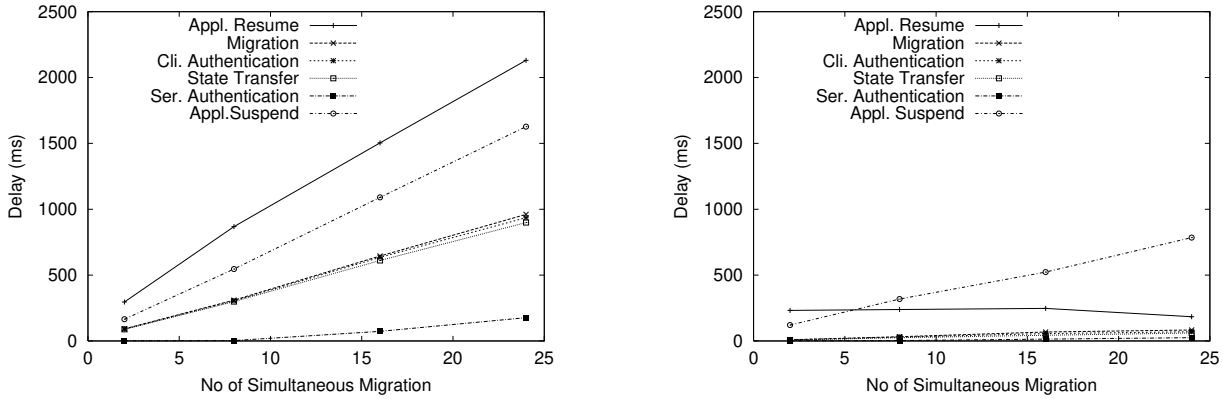
Fig. 17. Icecast: Same-host migration

sessions as the Icecast implementation leads to capacity related audio quality degradation and at times closure of the playback session under normal operation.

Figure 17(a) shows the average time spent in each stage of suspension and continuation with number of simultaneous sessions varying from 1 to 24. It is easy to observe that the application resumption time for Icecast sessions is very high compared to other applications. This is due to the inherent structure of the Icecast program and the flow continuation.

In the current modified Icecast implementation, the request for application suspension is serviced when the corresponding client is selected to be serviced. Hence, as the number of active play sessions increase, the average waiting time for selecting each client for transport increases thus affect our application suspension time. Similarly, new migrated clients are included for response once after every current client is serviced once (according to the Icecast implementation). This increases the overall latency of application resumption also. This high resumption latency is not caused by the Migrate service. The Icecast server without migration support would have had similar type of delay to introduce new clients under current implementation.

The time consumed in different stages of server migration for Icecast application are shown in 18(a). Apart of the application related overhead inherent in the Icecast application, the state transfer takes a significant time compared to the overall service latency. This is due to the fact that while a client session is suspended, the data buffer for that client is saved and transfered to the new location for playback. However, since Icecast is a real-time streaming service, the old data might not be useful at all. Hence, one way to reduce the service latency is by selectively storing the application data. Figures 17(b) and 18(b) show the latency when the client buffer data is not



(a) Icecast

(b) Icecast without buffers

Fig. 18. Icecast: Different-host migration

stored as part of the continuation. You can notice that the overall latency reduces significantly when the old data is not transfered to the new location. Finally, for client continuation as well as service continuation, the latency observed varies linearly with the number of simultaneous session suspension/resumption.

From all these experimental results, we conclude that the proposed migration-based continuation system does not introduce significant overhead limiting the scalability in terms of simultaneous resumption/suspension requests supported. It is also observed that the overall service latency depends on the application state information and can be reduced by judiciously selecting data to store in the continuation.

E. Discussion

We believe that continuation-based session migration and resumption is a powerful concept for enabling computing where communication connectivity is semi-permanent. Though it is used in this paper only to address the issue of network connectivity, the concept of continuation can be further exploited to account for semi-permanent availability of some other resources as well.

Generating the appropriate continuation is the central part of the proposed continuation-based session migration. As mentioned earlier in the paper, the two most important parts of generating appropriate continuation are: (1) finding out the location where the state of the application can be described concisely, otherwise called quiescent point, and (2) generating the appropriate continuation information for the used resources (network/file/terminal) so that session/application can be resumed efficiently. For the examples considered, it was very easy

to find the quiescent point as both applications followed the generic template of non-blocking select loops as a controlling location and a controlling thread to handle network events. We believe this step is going to be relatively easy for a programmer familiar with the overall architecture of the network application. However, generating appropriate continuation for non-network dependencies/resources is relatively harder. For example, in the SSHD example, in order to migrate sessions across hosts, appropriate information about the pseudo-terminals need to be saved in the continuation for the resumption to create appropriate pseudo-terminals. Apart from the resource dependencies, there are some application-specific dependencies which are difficult to capture. For example, X-forwarding with SSH uses the DISPLAY environment variable to tunnel X-related packets appropriately. When a session is moved to a different location, the DISPLAY variable needs to be changed appropriately for the session migration to be successful. Hence, in addition to using an inter-position agent to handle network related resource continuation automatically, one needs to think about alternative and simple ways to provide solutions for other resource-related continuation and application-dependencies. The application developer might be the best person to handle application-dependencies; it might be difficult for any other person to apprehend these dependencies.

The current version of Migrate using the TESLA wrapper is designed for non-threaded applications. In this system, we faced a few bottlenecks in enabling the session resumption/migration support for threaded applications like Icecast. First of all, in Red Hat Linux's thread implementation, we had a difficulty in using software interrupts to communicate events between TESLA and the applications. The problem lies in the interrupt implementation for threaded applications. We avoided the problem by putting all interrupt-related functionalities in one thread of execution. Another difficulty was due to the non-threaded architecture of TESLA. In the internal implementation of TESLA, process id and the file descriptor were being used as an index which is not valid for a threaded application where each thread has an unique process id. Additionally, the threaded design violated some of the assumptions made in TESLA about sequential operation in a process. Hence, we had to modify the synchronization step using an internal data structure to avoid the mismatch of access being made from different threads. It would be possible to support the threaded applications more efficiently if the TESLA library was redesigned to be thread-safe.

VII. Related work

Transparent approaches to handling mobility typically hide the disconnection or change in network location from the application by introducing a level of indirection. Proposals have suggested placing such indirection at various levels of the network protocol stack (including the network [19], transport [15], [23], [27], and session [13] layers), the filesystem (*e.g.*, Coda [16]), or even external to the operating system through virtual machine technology [12], [18], [21].

We are not the first to observe that applications may wish to control mobility event handling. Zhao [28] implemented a mechanism for individual applications to select among available network interfaces and mobility management schemes, but the available schemes offered either severely limited or entirely transparent mobility support. Collaborative system support for variable network conditions was pioneered by Odyssey [17], which introduced the notion of application-aware adaptation to bandwidth constraints.

A significant amount of research has focused on allowing mobile clients to continue to function while disconnected. In particular, applications based on the Remote Procedure Call (RPC) model, in which each communication is a request-reply exchange, have been successfully adapted for disconnected operation using the Rover toolkit [10], which queues RPCs for later delivery. Similar ideas were explored in Bayou [**?**], but both Bayou and Rover depart from the traditional, connection-oriented programming model for networked applications. In particular, applications are forced to deal with the notion of tentative transactions. Pervasive computing platforms like *One.world* [7] similarly require a complete redesign of existing applications.

Unmodified applications are traditionally suspended by creating snapshots, or *checkpoints*, of their process execution state, which can later be restored in order to resume process execution from the same state. This technique has been applied to migrate applications from one host to another or to restore applications after a system crash. A number of venerable research operating systems provided support for general process migration, including Sprite [5], V [3]. Unfortunately, due to its extreme complexity, process migration has found little success in commercial environments. Instead, user-level approaches have been proposed, most famously in Condor [14]. More recently, Zap [18] uses a notion of pods to migrate groups of related Linux processes between machines sharing an NFS file system. Unfortunately, many applications handle several sessions inside of one process; traditional process-based check-pointing does not

allow individual sessions to be independently suspended or resumed.

Other researchers have addressed fine-grained management of individual sessions within a process. A number of object migration systems [6], [7], [10], [11] have been proposed in the research literature, but they generally require a complete redesign of the application structure. For those applications well-suited to implementation in such a fashion, a number of commercial options exist as well. For example, the Java Servlet Specification supports the notion of explicitly storing application state inside session data structures, which can be individually encapsulated and shuttled between replica servers using the native Java serialization and RMI mechanisms. Servlet sessions include only application state, however, and do not reference any network connections or system resources (*e.g.*, files, locks, etc.) that may be needed.

A. Virtual machines

Recent proposals suggest suspending and resuming applications through a virtual machine monitor like VMware [12], [21]. Unless both end points are suspended simultaneously, however, an additional mechanism is necessary to preserve the session at the remote end point during suspension. Further, sessions can only be suspended and resumed independently if they run in separate VMs.

In comparison to session continuations, VM-based migration also suffers from a very long migration latency. Recently proposed optimizations [21] can reduce the state that needs to be transfered to resume the service, but it is still orders of magnitude larger than in our case. In particular, assuming a configuration of 256 MB of RAM and a 1-GB HDD as in the servers used for our evaluation, the latency of migration is of the order of 3 minutes, which is unbearable for service migration, especially when Icecast is used to stream live video.

B. Service Continuations

Perhaps the most similar system to ours was recently proposed by Sultan *et al.* [25]. Service continuations, as they are called, have similar aims, but the assumptions are different. For example, the service continuation assumes availability of cooperating servers for service to be migrated and migration policy is mostly implemented inside the operating system. In our system we provide for service migration in response to client mobility, with possible disconnected periods between suspend and resume. We assume the functionality of choosing the appropriate

server is better implemented outside the system using other lookup services available, hence cannot be known `a priori` for the support of server migration. Additionally, our approach does not assume any availability of services running currently to be able to migrate the server as long as the services can be resumed at the destination location. While migrating a particular service, the service may constitute many different network connections (especially in a gateway based deployment of service) hence require all the dependent connections to be migrate simultaneously for correct resumption of service. Migration of simultaneous connection migration efficiently can be supported very well in our system.

While both of the systems try to store the state of the application for use at the time of resumption, the state information is saved in our system on-demand based on disconnection or explicit request for migration. In the service-continuation based approach the state storage frequency needs to be higher, leading to a high overhead based on the amount of information that needs to be stored for any application.

VIII. Conclusion

Migrate's *session continuations* explicitly record all the state required for network applications to suspend sessions upon disconnection, migrate them to replica servers if desired, and properly resume them when connectivity is restored. Application programmers implement only a handler that generates the application-specific part of the continuation. When Migrate detects that a session has disconnected, it notifies the handler, which returns the corresponding continuation. When Migrate detects that a session's connectivity has been restored, it invokes the continuation (after possibly moving to a replica server). The Migrate service also virtualizes network connections to function across network address changes, verifies any host or system dependencies, allows portions of a process to be suspended and securely resumed, spawns new application processes if necessary, optimizes scarce system resources, and replicates any required system resources on replica servers.

We evaluated the effectiveness of Migrate's session continuation approach by studying how SSH, and Icecast can be extended to implement a suspend/resume facility. The required source code changes are small—between 0.5% and 1.75% of the total—and the generated continuations are only between 1% and 5% of the process' memory footprint. Migrate is also effective in conserving resources of suspended sessions, such as open file descriptors. Finally, measurements

with Migrate's implementation on Linux showed that session resumption times are a few hundred milliseconds, *i.e.*, small compared to typical disconnection durations.

These results together indicate that session continuations provide an easy-to-use and effective abstraction for Internet servers that wish to migrate client sessions across replicas. Migrate's session continuations have the potential to improve the experience of mobile users by simplifying the job of application designers. We believe that, in many cases, it may be easier for programmers to use Migrate's facilities rather than redesign their session-oriented application in a transactional fashion. While our initial results with SSH and Icecast are promising, further study of additional session-based applications—including those that have already been implemented in a stateless manner—is necessary in order to verify this claim.

REFERENCES

[1] A. W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[2] H. Balakrishnan, H. Rahul, and S. Seshan. An integrated congestion management architecture for Internet hosts. In *Proc. ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, pages 175–187, Cambridge, Massachusetts, Aug. 1999.

[3] D. R. Cheriton. The V distributed system. *Communications of the ACM*, 31(3):314–333, Mar. 1988.

[4] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-11:644–654, Nov. 1976.

[5] F. Douglis and J. Ousterhout. Transparent process migration: Design alternatives and the Sprite implementation. *Software - Practice and Experience*, 21(8):757–785, Aug. 1991.

[6] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal on Supercomputer Applications*, 11(2):115–128, 1997.

[7] R. Grimm, J. Davis, E. Lemar, A. Macbeth, S. Swanson, T. Anderson, B. Bershad, G. Borriello, S. Gribble, and D. Wetherall. System support for pervasive applications. *ACM Transactions on Computer Systems*, 2004. To appear.

[8] Icecast Project. Icecast streaming server. `http://www.icecast.org`.

[9] J. Inouye, J. Binkley, and J. Walpole. Dynamic network reconfiguration support for mobile computers. In *Proc. 3rd Annual ACM/IEEE International Conference on Mobile Computing and Networking*, pages 13–22, Budapest, Hungary, Sept. 1997.

[10] A. D. Joseph, J. A. Tauber, and M. F. Kaashoek. Mobile computing with the Rover toolkit. *IEEE Transactions on Computers*, 46(3):337–352, Mar. 1997.

[11] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, Feb. 1988.

[12] M. A. Kozuch and M. Satyanarayanan. Internet suspend/resume. In *Proc. IEEE WMCSA*, pages 40–46, Callicoon, NY, June 2002.

[13] B. Landfeldt, T. Larsson, Y. Ismailov, and A. Seneviratne. SLM, a framework for session layer mobility management. In *Proc. IEEE International Conference on Computer Communications and Networks*, pages 452–456, Natick, Massachusetts, Oct. 1999.

[14] M. Litzkow, M. Livny, and M. Mutka. Condor — A hunter of idle workstations. In *Proc. 8th International Conference on Distributed Computing Systems*, pages 104–111, San Jose, California, June 1988.

[15] D. Maltz and P. Bhagwat. MSOCKS: An architecture for transport layer mobility. In *Proc. IEEE Infocom*, pages 1037–1045, San Francisco, California, Mar. 1998.

[16] L. Mummert, M. Ebling, and M. Satyanarayanan. Exploiting weak connectivity for mobile file access. In *Proc. 15th ACM Symposium on Operating Systems Principles*, pages 143–155, Copper Mountain, Colorado, Dec. 1995.

[17] B. Noble, M. Satyanarayanan, D. Narayanan, J. E. Tilton, J. Flinn, and K. R. Walker. Agile application-aware adaptation for mobility. In *Proc. 16th ACM Symposium on Operating Systems Principles*, pages 276–287, Saint Malo, France, Oct. 1997.

[18] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The design and implementation of Zap: A system for migrating computing environments. In *Proc. 5th USENIX Symposium on Operating Systems Design and Implementation*, pages 361–376, Boston, Massachusetts, Dec. 2002.

[19] C. E. Perkins. IP mobility support for IPv4. RFC 3220, Internet Engineering Task Force, Jan. 2002.

[20] J. C. Reynolds. The discoveries of continuations. *LISP and Symbolic Computation: An International Journal*, 6:233–247, 1993.

[21] C. P. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. S. Lam, and M. Rosenblum. Optimizing the migration of virtual computers. In *Proc. 5th USENIX Symposium on Operating Systems Design and Implementation*, pages 377–390, Boston, Massachusetts, Dec. 2002.

[22] A. C. Snoeren. *A Session-Based Approach to Internet Mobility*. PhD thesis, Massachusetts Institute of Technology, Dec. 2002.

[23] A. C. Snoeren and H. Balakrishnan. An end-to-end approach to host mobility. In *Proceedings of the ACM MOBICOM Conference*, pages 155–166, Boston, MA, Aug. 2000.

[24] G. Su and J. Nieh. Mobile communications with virtual network address translation. CUCS-003-02, Columbia University, Feb. 2002.

[25] F. Sultan, A. Bohra, and L. Iftode. Service continuations: An operating system mechanism for dynamic migration of Internet service sessions. In *Proc. 22nd Symposium on Reliable Distributed Systems*, Florence, Italy, Oct. 2003.

[26] T. Ylonen, T. Kivinen, T. J. Rinne, and S. Lehtinen. SSH protocol architecture. Internet Draft, Internet Engineering Task Force, Jan. 2001. `draft-ietf-secsh-architecture-12.txt` (work in progress).

[27] V. C. Zandy and B. P. Miller. Reliable network connections. In *Proc. 8th Annual ACM/IEEE International Conference on Mobile Computing and Networking*, pages 95–106, Atlanta, Georgia, Sept. 2002.

[28] X. Zhao, C. Castelluccia, and M. Baker. Flexible network support for mobile hosts. *ACM Mobile Networks and Application Journal*, 6(2):137–149, Apr. 2001.