Enabling Internet Suspend/Resume with Session Continuations

Alex C. Snoeren SNOEREN@LCS.MIT.EDU
MIT Lab for Computer Science, 545 Technology Square, Cambridge, MA 02139

1. Introduction

Mobile laptop users have grown accustomed to the "suspend/resume" model of computing, in which activity can be resumed precisely from the point at which it was suspended, despite arbitrary periods of inactivity. Unfortunately, today's Internet hosts lack support for seamless operation of session-based network applications across periods of disconnectivity; hence, contemporary operating systems do not provide "suspend/resume" support for such applications. Instead, movement or disconnection events are either concealed inside the network or exposed as communication failures to the application, which is then forced to abandon open sessions and begin new ones.

The first approach is problematic to implement, often yields sub-optimal performance, and results in significant amounts of wasted resources. In particular, concealing disconnectivity (Zhang & Dao, 1995) becomes extremely difficult when applications require periodic communication or keep-alive messages. Further, if concealment is successful, the application is unable to adapt to changes in network conditions and continues consuming system resources (CPU, memory, kernel buffers, timers, file descriptors, etc.) while disconnected. Many of these resources are scarce, and cannot be efficiently multiplexed.

The latter, more common approach, typically results in a poor user experience, decreased performance, and even loss of data or incorrect operation unless the application is specifically designed to be restartable. In the best case, the user is forced to restart her application session, an inconvenient and (relatively) slow process, and often results in the loss of unsaved data–sometimes irretrievably (e.g. live streaming media). In more unfortunate situations, the inability to communicate with the remote host for some period of time may lead to the invocation of an undesirable operation by the remote host.

In both cases, the problems arise from the inability to effectively manage session state across suspend/resume events. In order to resume communication sessions from the point of disconnection, communicating hosts must record the session state to be restored and reconciled with the resumption environment, and various network-dependent resources reconfigured (such as transport protocols, name bindings, application settings, etc.)

We observe that the resumption problem is not unlike that encountered by a compiler when handling returns from procedure calls. In both cases, naming scopes, environment settings, and mutable state must be saved and restored. This can all be avoided if procedures never return, which lead to the development of *continuations*, which embody the "the rest of the computation." Rather than returning, procedures can simply execute a provided continuation, resulting in a chain of procedure calls that never return (Appel, 1992).

In a similar spirit, we propose *session continuations*, which allow application sessions to suspend operation during periods of disconnection and specify their resumption context, enabling the release of unnecessary resources and intelligent adaptation to the reconnection environment. Session continuations provide application programmers with a single abstraction that is at once simple to program for and powerful enough to enable sophisticated resource savings. We have implemented session continuations as part of the *Migrate* mobility toolkit (Snoeren et al., 2001) to manage both scarce system resources, such as kernel socket buffers, memory, and file descriptors, as well as application state across changes in network attachment point and during periods of disconnectivity. Here, we demonstrate how session continuations can conserve system resources by describing an extended SSH server that consumes almost no resources on the server when the client is disconnected, yet enables seamless session resumption upon reconnection.

2. Session continuation passing

Continuation Passing Style (CPS), where the thread of control is explicitly passed from one continuation to another along with any necessary context, has been shown to be beneficial in several domains, including process management and IPC (Draves et al., 1991). The key advantage of CPS is that any state or context necessary for the continuation is specified explicitly, and control never returns to the entity calling the continuation. This significantly simplifies state management, as context must never be transparently saved and restored—all durable state is explicit.

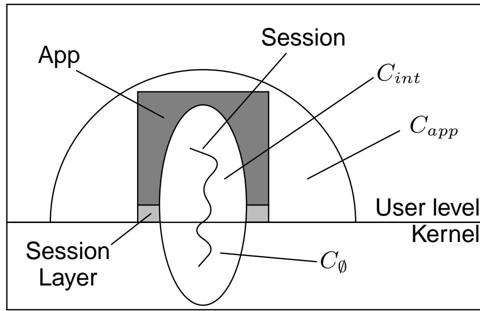

Figure 1. Three separate continuations make up a complete session-continuation: a base continuation, $C_\emptyset$, an internal continuation C_{int}, and one that restarts the entire application, C_{app}.

By making the notion of the "rest of the session" explicit, session continuations similarly enable graceful handling of session disconnection, reconnection, and rebinding. Upon disconnection, each session end point simply generates a continuation representing the state necessary to complete the session. With the continuation safely stored, the previously communicating hosts are free to reclaim any resources used by the abandoned session. If a host ever wishes to resume the session, each system need only invoke their continuation to continue processing.

Note that a session continuation is *not* simply a snapshot of the current local state; rather, it is a function from a quiescent state (as determined by the application) to the remainder of the session. In order to adapt to dynamic conditions, continuations take a set of input parameters reflecting both the current network state and optionally that of the remote end points (as exchanged during reconnection).

The contents of a continuation depend greatly on the state of the system. For example, if a continuation will execute in the same process that generated it, and no associated state has changed during disconnection, the continuation simply needs to validate and perhaps update the end-point bindings (since the corresponding host may have changed network locations) and identify the point in the application to which control should be returned. We call this simple re-binding continuation the base continuation, $C_\emptyset$.

More generally, changes will have occurred in both local and remote state. In many cases, the session continuation will execute in the same process it was created in (server applications often host several client sessions per process). Internal continuations, $C_{int}()$, need only contain sufficient application state to allow the session to continue, and a function that restores the context required by the session. Sometimes, the application itself may need to be restarted; not only must system resources be established appropriately, but the appropriate application must be continued (through its own continuation, $C_{app}()$) before the application-specific continuation can be run. This composition of continuations is depicted in Figure 1.

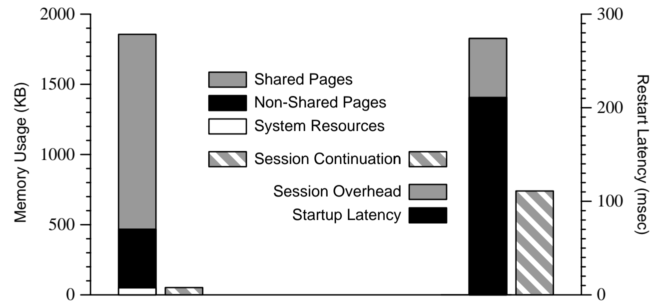

Figure 2. Through the use of session continuations, our SSH server is able to dramatically reduce its resource footprint during periods of disconnection. In addition, session continuation is significantly faster than session initiation.

3. SSH: A case study

To demonstrate the ability of session continuations to conserve system resources, we have extended an SSH server to support session continuations. In only 250 lines of code[1] and an afternoon's worth of work, we were able to extend SSH to generate both internal and application continuations. If a client unexpectedly becomes disconnected, the SSH server application can remove itself completely from the server by presenting a 52KB session continuation to *Migrate* which is cached in secondary storage. As can be seen in Figure 2, this frees up 1856KB of memory (468KB of which was not shared) on the server. More importantly, however, it also releases system resources including network connections, kernel buffers, and file descriptors. Upon reconnection, the continuation resumes execution within 111ms—significantly less than the 274ms required to initiate the session (although 63ms of that is additional cryptography overhead due to the *Migrate* mobility support). Hence, the resulting application provides seamless "suspend/resume" operation to clients with negligible resource usage during periods of disconnection.

References

Appel, A. (1992). *Compiling with continuations*. Cambridge University Press.

Draves, R. P., et al. (1991). Using continuations to implement thread management and communication in operating systems. *Proc. ACM SOSP* (pp. 122–136).

Snoeren, A. C., Balakrishnan, H., & Kaashoek, M. F. (2001). Reconsidering Internet mobility. *Proc. HotOS VIII* (pp. 41–46).

Zhang, Y., & Dao, S. (1995). A "persistent connection" model for mobile and distributed systems. *Proc. IEEE ICCCN* (pp. 300–307).

[1]The SSH server itself contains approximately $48,000$ LOC.