SAS Kernel: Streaming as a Service Kernel for Correlated Multi-Streaming

Pooja Agarwal, Raoul Rivas, Wanmin Wu, Ahsan Arefin, Zixia Huang, Klara Nahrstedt
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois, United States
{pagarwl, trivas, wwu23, marefin2, zhuang21, klara}@illinois.edu

ABSTRACT

This paper presents a novel paradigm of *Streaming as a Service (SAS)* to model correlated multi-streaming in Distributed Interactive Multimedia Environments. We propose *SAS Kernel*, a generic, distributed, and modular service kernel realizing SAS concept. SAS Kernel features high flexibility by employing a configurable interface to allow for input of correlated multi-streams (bundle of streams) from diverse types of sensory devices. It is also highly extensible by allowing user-controlled functions to be applied to bundle of streams in runtime. Experiments with real-world applications demonstrate that the SAS Kernel incurs low overhead in delay, CPU, and bandwidth demands.

Categories and Subject Descriptors

C.2.4 [**Computer Communication Networks**]: Distributed Systems—*Network Operating System*; D.4.7 [**Operating Systems**]: Organization and Design

General Terms

Design, Measurement, Performance

Keywords

Streaming as a Service, SAS Kernel, End Device Abstraction

1. INTRODUCTION

Cyber-Physical systems using large number of sensors are fast becoming ubiquitous. An example of multi-sensory system is Distributed Interactive Multimedia Environments (DIMEs). DIMEs allow real-time collaborative activities like interactive gaming, physical therapy, and sport activities across multiple, geographically distributed users. Some of the real applications include Physical Training [5], Virtual Gaming [16], and Teleimmersive Dancing [11].

DIMEs are comprised of input devices (e.g., cameras, microphones, body sensors, haptic devices) and output devices (e.g., displays, speakers, actuators). DIMEs make use of service gateways to transfer content from input devices to remote output devices over the Internet. Apart from streaming, DIME service gateways need to provide various functionalities like overlay routing, bandwidth management, QoS provisioning, synchronization, and monitoring.

Several architectures for streaming gateways interconnecting local area networks like Ethernet, Bluetooth, wireless access to the Internet exist in the literature. However, these service gateways [9], [10], [13], [14], [15] are limited to functionalities like relaying, multiplexing, translating, and managing local resources only, which fail to satisfy the requirements of DIMEs. There are some proprietary gateways developed at HP Halo, Cisco Telepresence, and Technicolor, however they are tailored to cater closed applications and their internal functionalities are publicly unknown.

DIME requirements differ from those of traditional gateways as:

1. Presence of multiple correlated sensors interacting in a DIME session requires support for *large scale correlated multi-streaming* as an inherent functionality.
2. High interactivity in DIME sessions requires *soft real-time delivery* of all streams.
3. *Advanced QoS services* across multiple spatially and temporally correlated streams need to be supported.
4. With end-devices dispersed across remote locations, *distributed resource management* is important in DIMEs.
5. Lack of standard stream formats across sensors from diverse vendors require *unified interface* with the end-devices.

Based on the above requirements, concepts of *correlated multi-streaming*, *device abstraction*, and *user-controlled services* need to be supported in DIMEs. Other important challenges include *flexibility*, and *scalability* of DIME frameworks.

In this paper, we envision a novel paradigm, *Streaming as a Service (SAS)* to model correlated multi-streaming service, where correlated multi-streams, also called *bundle of streams* are first class objects. We propose a SAS-based, generalized, distributed service kernel, *SAS Kernel* to setup, process, and control bundles of streams. We emphasize that SAS and SAS Kernel are not limited to just DIMEs but provide a fundamental foundation of modern service-oriented architectures for wide range of stream-based applications (e.g., 3D Streaming).

In summary our contributions in this paper are:

1. Formalization of the SAS paradigm (see Section 2);
2. Design of the SAS Kernel with properties:
   (a) Streams and bundles as first class objects (Section 3),
   (b) Unified interface for diverse end-devices (Section 5.1),
   (c) User-controlled runtime functions over streams and bundles (Section 3 and 5.2),
   (d) Integrated session and bundle management (Section 3),
   (e) Extensibility of SAS components (Section 5)
3. Quantitative evaluation in a real testbed (Section 6).

NOSSDAV'11, June 1–3, 2011, Vancouver, British Columbia, Canada.

2. STREAMING AS A SERVICE (SAS)

We propose a generalized Streaming as a Service paradigm for platforms providing correlated soft real-time multi-streaming as the key service. The guiding properties of SAS are as follows:

- *Distributed Correlated Multi-Stream Support* - DIMEs are composed of distributed correlated multi-streams called *Bundle of Streams (BoS)* [1] sharing high spatial and temporal correlations. These bundles interact in synchronous and soft real-time manner. The current streaming protocols like RTP/RTCP, SIP, RTSP do not take into account efficiently the spatio-temporal dependencies among large sets of streams. To deal with this, the SAS model inherently supports: (1) large scale correlated soft real-time streaming, (2) end-to-end session management based on media correlations [8].

- *Universal Open Access* - Unlike the Internet Protocols which have become the lingua franca, there is a lack of well-agreed formats across emerging devices like 3D cameras and microphone arrays. To overcome the problem of implementing large sets of formats, SAS model supports universal access policy with well-defined interfaces to the end-devices. In SAS, varied types of streaming devices with different standards seemlessly connect and stream the data.

- *User-defined Functions* - SAS provides the flexibility of provisioning two types of run-time functions on streams: (1) system-defined functions like rate control, congestion control, and multi-stream synchronization, (2) user-defined functions like compression, encryption, and view management. These functions can be requested in an on-demand basis.

- *Availability* - It is anticipated that in the future, access to SAS will follow "always on" paradigm, like cable modem access is today. Thus, SAS is highly available at all times.

- *Robustness* - For SAS, the capability to monitor performance, isolate faults, and automatically recover from faults is critical. The robustness of SAS comes through (1) On-demand monitoring services with varied resolutions, (2) Fault localization and easy recovery mechanism.

- *Scalability* - It is anticipated that larger scale of sensors will enhance the Quality of Experience (QoE) of users. Thus, SAS provides scalability in terms of supporting large sets of streams. Also, extensibility of a SAS-based framework is important to support adding new functional services as need arises.

Thus, the goal of the SAS is to foster bundles of streams needing correlated multi-streaming support, universal access across diverse devices, and user-controlled runtime functions in future DIME systems. To realize the SAS paradigm, we present SAS Kernel, set of real-time integrated services that enforce *SAS properties* (as outlined above) at runtime.

3. SAS KERNEL FRAMEWORK

In DIMEs, distributed end-devices share streams and resources in real-time collaborative sessions. The SAS Kernel provides runtime system for easy setup, processing, management, and access of bundles of streams. The SAS Kernel implements the SAS properties as follows:

Strong distributed correlated multi-streaming support: SAS Kernel ensures correlated multi-streaming by (1) Managing and keeping states of streams, bundles, sessions, and resources, (2) Providing streaming policies for correlated soft real-time scheduling, co-operative congestion control, and overlay routing. The Management Entities (Figure 1) handle correlated multi-streaming.

Universal access and easy availability: SAS Kernel provides (1) Unified interface for end-devices, (2) Easily configurable stream specifications to describe device and stream characteristics. The SAS Interface (Section 5.1) provides universal access.

Runtime mechanisms and functions: (1) SAS Kernel follows the principle of *Separation of mechanism and policy* [6], i.e., the mechanisms only provide a unified framework for plugging-in the policies/functions and the actual functions are implemented at the user space, (2) SAS Kernel provides runtime loading of *user-defined functions* operating over streams or bundles. The Runtime Entities (Figure 1) and Function Manager (Section 5.2) provide functions and mechanisms. This design also allows easy availability.

Robustness: To ensure robustness (1) A cross-layer online monitoring interface is provided, (2) Runtime analysis of monitoring data to trigger recovery procedures is done. The Monitoring Manager (Figure 1) ensures robustness.

Scalability: SAS Kernel provides both device scalability and stream service extensibility via (1) User-configurable interfaces with varied end-devices, (2) Modular design of stream services to allow user-defined functions on streams, (3) Modular design of all SAS Kernel entities built on the principle of separation of concerns. The SAS Interface and Function Manager (Section 5) discuss the extensibility of the framework.

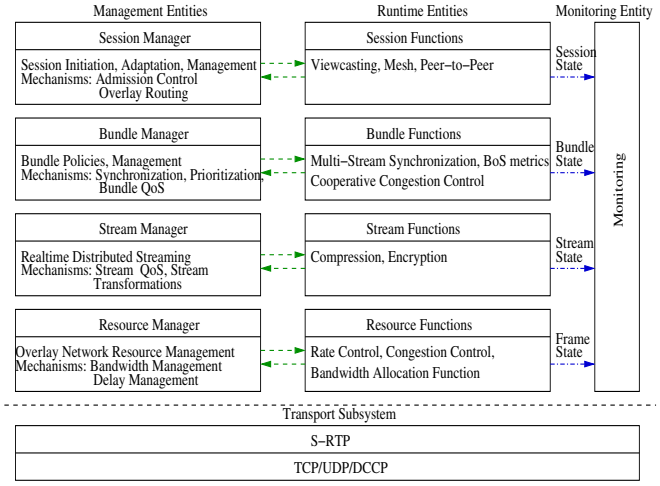


Figure 1: SAS Kernel Framework

In SAS Kernel, the SAS properties get implemented as management, runtime, and monitoring entities in the session subsystem on top of a transport subsystem. Since streams are first class objects in SAS Kernel, each of the entities keeps track of and controls streams and stream derivatives (e.g., bundles, frames). Figure 1 shows the layout of various entities over the transport subsystem. A brief description of each of them is as follows:

Management Entities: The management entities manage sessions, bundles, streams, frames, and their corresponding resources. They provide mechanisms for generic tasks like overlay routing and provide interfaces to dynamically load the runtime entities. There are four management entities:

1. Session Manager - It performs session initiation, membership control, and session management. It takes management decisions and provides mechanisms to load session level functions like overlay routing and admission control.

2. Bundle Manager - It handles the correlation between the streams and defines the policies to group multiple streams into correlated bundles of streams. It provides mechanisms for runtime functions over these correlated bundles of streams like cooperative congestion control, prioritization, view management, and bundle of streams (BoS) metrics [1].
3. Stream Manager - It keeps states about receipt and delivery of streams across sites and determines policies for streaming. It categorizes streams as InStreams (from input devices) and OutStreams (to output devices). Mechanisms for stream-based runtime functions like compression, encryption are also provided by this manager.
4. Resource Manager - It manages overlay network resources like bandwidth and delay to ensure real-time delivery of streams.

Runtime Entities: The runtime entities provide specific system/user-defined policies for the mechanisms like Mesh protocol for overlay routing. These entities are dynamically pluggable real-time functions operating over sessions, bundles, streams, frames, and network resources. These entities are open to be either implemented by SAS Kernel system-admins or the end-users. Examples of runtime entities at each level are shown in Figure 1.

Monitoring Entity: SAS Kernel implements a cross-layer event-driven monitoring entity. This entity provides real-time monitoring plane for overall system monitoring. The monitoring entity forms a feedback loop by communicating the states from the run-time functions to the corresponding managers, allowing the managers to take appropriate actions like adaptation, or policy switching. The monitoring entity also monitors for faults and failures.

Transport Subsystem: To ensure soft real-time delivery, the transport subsystem abstracts the underlying transport layer protocols allowing end-users to dynamically request appropriate protocols like TCP, UDP, DCCP based on application type and network conditions. The frames are encapsulated using our DIME specific S-RTP protocol (section 5.1.3) which adds semantic information (used by managers) like stream type, functions requested, device addressing, and streams in same bundle.

4. STREAM FLOW IN SAS KERNEL

Distributed SAS Kernel is realized through a set of multiple distributed SAS gateways and SAS interfaces as shown in Figure 2. SAS gateways take on the responsibility of hosting the SAS Kernel instances and the SAS interfaces (SASI) provide the connectivity between the end-devices and the SAS Kernel. We assume that all gateways and end-devices can be connected to each other via the Internet. Figure 3 shows the end-devices, SAS interfaces, and the functional placement of the SAS Kernel entities in a gateway. The streaming algorithm is as follows:

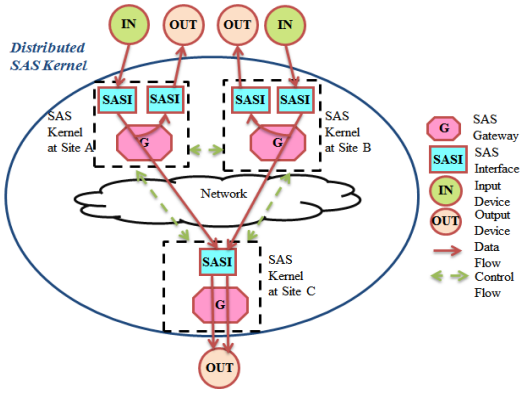


Figure 2: Distributed SAS Kernel Components

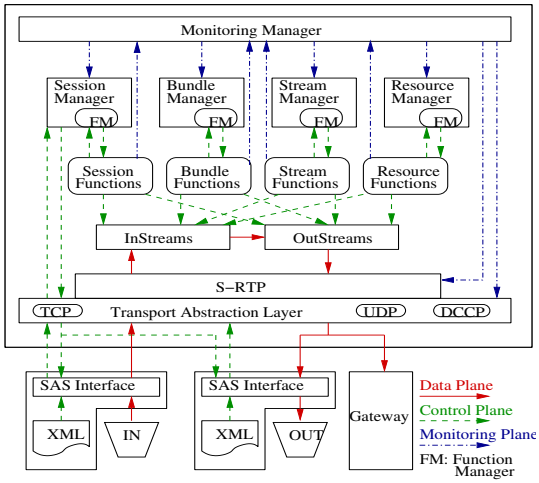


Figure 3: SAS Kernel Data, Control, and Monitoring Planes

1. Session Initiation: A streaming end-device first starts a connection with the SAS interface present at the end-device machine. The SAS interface initiates a session with the closest SAS gateway and requests the services specified in the user-defined XML configuration. The request is handled by the Session Manager in the gateway. It verifies if the requested services are supported and sends an *ACK* to the SAS interface. On positive *ACK*, Session Manager opens data and control connections with the end-device through the SAS interface. It also constructs overlay routing topology with other gateways, stores the meta-data about the new session, instantiates a Stream Manager for the joined stream, groups streams into bundles, and instantiates Bundle Manager.
2. End-to-End Streaming: An input device communicates its stream to the SAS interface. The SAS interface applies the S-RTP headers on each packet based on the information specified in the XML file. The packets are then sent over a chosen transport layer protocol to the corresponding InStream instantiated by the Stream Manager for this session. Once the InStream starts to get delivered in SAS Gateway, the Stream Manager creates corresponding sets of OutStreams based on number of requesting output devices. The InStreams are then connected to the respective OutStreams.
3. Runtime Functions: The run-time functions are loaded by the Function Manager (FM) present in each of the Managers. The InStreams and OutStreams are processed through the Bundle Manager to apply user-demanded bundle functions over bundles. The Stream Manager then applies stream based functions. For resource optimization, Resource Manager applies policies for bandwidth management and congestion control. It must be noted that streams pass through all these functions only when the user demands them. Thus, no extra overhead is incurred unless some functions are specified. This ensures fastest delivery of streams.
4. Monitoring: Each entity implements hooks and callbacks to send monitoring information like QoS performance, resource utilization, and faults to the Monitoring Manager. Based on the received information, Monitoring Manager takes appropriate QoS or fault tolerance measures.

5. SAS KERNEL DESIGN

The two main components of SAS Kernel are SAS Interface and Kernel Function Managers which are discussed in detail in the following subsections.

5.1 SAS Interface

The device-SAS Kernel interface provides universal open access (section 3) and faces the challenges of (1) Multiple non-standardized stream formats of end-devices, (2) Requirement to understand all the stream formats to allow functions over streams.

The above challenges severely affect the scability and flexibility of the service gateways. To address this issue, current solutions only implement a subset of these stream formats and thus, fail to support devices from diverse vendors. Instead, our approach relies on separating the stream formats from the main SAS Kernel using configuration mechanisms to specify the formats at runtime. Thus, SAS Kernel realizes four concepts: 1) End-to-End Tunneling, 2) Device Stream Specification, 3) Semantic data propagation through S-RTP, and 4) Service Negotiation.

5.1.1 End-to-End Tunneling

The idea behind SAS Kernel is that end-devices should interact agnostic of the SAS Kernel i.e. the end-devices do not know if they are communicating via SAS Kernel. The challenge in providing agnostic connection is that there should be *no source code modification at the end-devices*. To achieve this, POSIX socket API is used as an interface between end-devices and SAS Kernel.

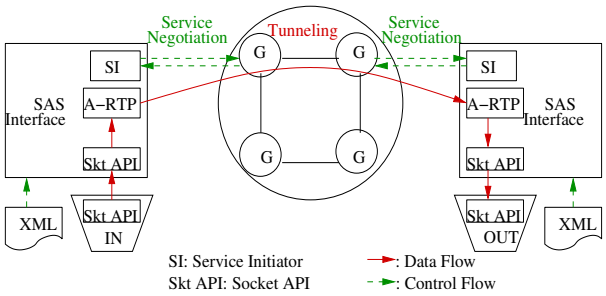


Figure 4: Socket Interface and Tunneling

The assumption behind using socket API is that the end-devices in DIMEs mostly follow client-server type of connections and they usually provide interface to specify the IP and port number of the remote device. Thus, the end-devices can be dynamically configured to connect to the SAS interface. The SAS interface, placed at each device, uses socket API to intercept the traffic from the input devices and send it via the SAS Kernel to the output devices. In addition, a peer-to-peer virtual tunnel is created between the devices where the virtual tunnel is supported by the underlying SAS Kernel. Figure 4 shows the socket interface and the tunnel.

5.1.2 Device Stream Specification

In order to apply functions on streams, SAS Kernel needs to understand the semantics of the stream, i.e. the packet structure. Thus, SAS interface requires end-users to provide a simple high-level specification of the stream semantics in a user readable language like XML. The specification is composed of two main parts: (1) Device Specification containing general metadata about the device, (2) Stream Specification containing stream format.

The Device Specification consists of a unique identifier for addressing the device in the originating site, the content type (e.g., video, audio), content subtype (e.g., for video point cloud, mesh) and the transport protocol the device uses (e.g., TCP, UDP). The Stream Specification specifies the format of the sequence of data packets as they appear within the stream. There are two general formats: fixed-size packets and variable-size packets. The fixed-size packets require only packet size to be specified while the variable-size packets require a fixed size header containing the packet size to

```
<DEVICE SPECIFICATION>
  ...
   <PROTOCOL_TYPE> TCP </PROTOCOL_TYPE>
   <TYPE>VIDEO </TYPE>
   <SUBTYPE> POINT-CLOUD </SUBTYPE>
</DEVICE SPECIFICATION>
<STREAM SPECIFICATION>
   <PACKET_FIXED>
      <HANDSHAKE> ON </HANDSHAKE>
      <PACKET_SIZE> 140 </PACKET_SIZE>
      <PACKET_COUNT> 1 </PACKET_COUNT>
   </PACKET_FIXED>
   <PACKET_VARIABLE>
      <HANDSHAKE> OFF </HANDSHAKE>
      <HEADER_SIZE> 10 </HEADER_SIZE>
      <HEADER_OFFSET> 6 </HEADER_OFFSET>
      <DATASIZE_TYPE> 4</DATASIZE_TYPE>
      <PACKET_COUNT> -1 </PACKET_COUNT>
   </PACKET_VARIABLE>
</STREAM SPECIFICATION>
```

Figure 5: Device Stream Specification for a video stream

be specified. Other stream parameters like frame rate, color information are specified through *Handshake packets* between the end-devices. This specification allows for marking packets as *Handshake packets*. SAS Kernel forms a multicast network between the input devices and the output devices, requiring storing and replaying these *Handshake packets* when new output devices are added to the kernel. The packet count specifies how many of each type of packets are present consecutively in the stream.

Figure 5 shows an example XML configuration file used in the 3D Tele-immersion system in our lab for a video stream. The camera protocol is comprised of single fixed handshake packet of 140 bytes followed by all (packet count of -1 indicates possibly infinite) payload packets of variable size that have a header of 10 bytes, with packet size specified at byte 6 in the header. Moreover, this specification is easy to implement and flexible enough to allow a wide range of end-devices to interface with our SAS interface without modification or recompilation.

5.1.3 SAS Real-Time Protocol (S-RTP)

Each data packet read by the SAS interface is then encapsulated using the SAS Real-Time Protocol. S-RTP is similar to RTP but it is tailored to include DIME specific session semantics and lighter-weight. Through S-RTP, session semantics like device addressing, services requested, and groups of streams forming bundles are marked on each packet, allowing easy dissemination of each stream's state to all SAS components.

Bit Offset	0–31	32–47	48–63	64–95	96–127
0	Version	SID	RID	DID	TOS
128	Stream Type	Stream SubType		Frame Timestamp	
256	<BundleList>			Header	Payload(Variable)
256+64*C+64	... (Variable Payload)			Frame Number	

Figure 6: S-RTP Header Specification

The structure of the S-RTP packet is shown in Figure 6. The packet first specifies the version of the S-RTP protocol followed by a 64 bit unique stream identifier. The unique identifier uses a hierarchical addressing scheme composed of the DIME session ID (SID), the DIME site/room identifier (RID), and the device identifier (DID) within the room. The Type of Service (TOS), a 64 bit flag vector, specifies the requested functions, the state information about functions that were applied along the route in SAS Kernel, and a Handshake bit to specify Handshake packet.

The stream type and subtype together form a tuple to uniquely identify the type (video, audio, sensory data) and the data format (e.g. for video, mesh and point-cloud). Next, S-RTP packet con-

tains a list of all stream IDs forming a bundle <BundleList>, timestamp of packet creation, fixed/variable payload, and frame number.

5.1.4 Service Initiation and Negotiation

After reading the stream specification and constructing an S-RTP packet, the SAS interface at the joining end-device initiates a session with the SAS Kernel. The Session Manager in the SAS Kernel handles the session initiation and service negotiation tasks. Remote procedure calls (RPC) and marshalling is used between SAS interface and SAS Kernel and simple session initiation and negotiation protocols are used as can be found in the literature. Our contribution in SAS Kernel is that the SAS Kernel allows dynamic pluggability of different correlations based admission control and bundle routing algorithms as need arises in the session and resource management.

The SAS interface sends a *JOIN* request message specifying desired transport protocol to use, the characteristics of the bundles and joining streams (e.g., periodic or aperiodic, variable or fixed packet sizes, payload type, payload sub-type, expected bandwidth usage), and the services requested (encryption, compression, congestion control). Upon receipt of the *JOIN* message, the SAS Kernel verifies whether it can support services requested, and if so, opens required data ports and returns an *ACK* containing the ports. The SAS Kernel renegotiates if it does not support any of the services with the SAS interface. The Session Manager in SAS Kernel then creates InStreams and Bundles accordingly, bookmarks the parameters, and uses the data channels for data transfer.

In case of output end-device join, the payload type and subtype tuple provides a hierarchical way for the SAS Kernel to determine which bundles should be routed to the output device by matching the payload type and sub-type of the possessed InStreams with those specified. For example, one may use two renderers to display the frontal and back camera streams respectively; although they all identify the "video" type, one renderer and the frontal cameras use the "frontal" sub-type, and the other renderer and the back cameras use the "back" sub-type.

The strategy for interconnection and exchange of streams between the SAS gateways depends on the chosen routing protocol in the SAS Kernel. Some useful routing protocols for DIMEs are application level multicasting [17] like Viewcasting, Mesh. The discussion of these routing algorithms is out of scope of this paper.

5.2 SAS Kernel Function Manager

In order to provide runtime stream-processing functions, i.e., user controllable functions (as discussed in section 3), each manager in SAS Kernel implements a Function Manager (FM) as shown in Figure 3. The function manager is responsible for: (1) Implementing mechanisms, and (2) Scheduling functions on bundles, streams, and frames.

To support extensible operations, SAS Kernel divides the execution plane in two spaces: *End-User Space* and *System Space*. User-controllable functions are in the End-User Space while all the other SAS Kernel functions and resource management remain in the System Space. New functions to be added to the SAS Kernel are compiled separately by end-users into dynamically linked libraries and these functions are loaded and linked at runtime by the FM. Functions interact with FM using system calls (Syscalls) and FM uses Upcalls to the functions. Figure 7 shows FM architecture.

The Syscalls provide direct access to the bundle and stream metadata, S-RTP packet format, and also to the raw payload implemented by the end-devices. Each function implements an object and FM keeps the state information, allocates memory and forks threads. This makes FM suitable for supporting parallel concurrent

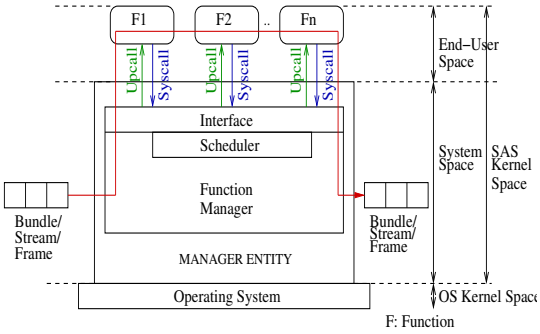


Figure 7: SAS Kernel Function Manager

functions. Functions are executed as a computing pipeline where the user can configure the order in which the operations are applied. A scheduler inside FM is responsible of context switching to the corresponding operation. FM thus provides the support for defining mechanisms at each level of data abstraction and load user-specific functions to implement these mechanisms. This ensures high extensibility of services in SAS Kernel.

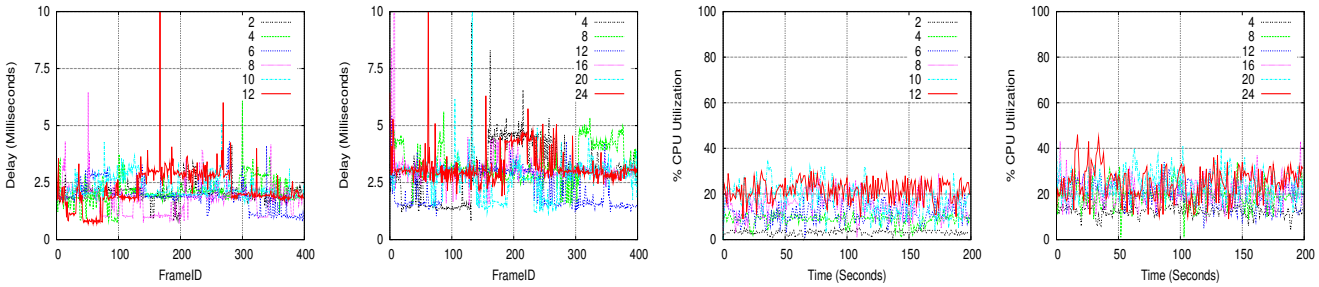
6. EVALUATION

We evaluate the performance of SAS Kernel in a real 3D Teleimmersion (3DTI) DIME System at University of Illinois, Urbana Champaign. 3DTI typically includes 4 to 5 sites, each producing 4 to 5 streams comprising of 3D cameras and microphones. Each site spawns a gateway for the SAS Kernel. Each stereo camera produces a variable 3D video stream ranging between 6 to 10Mbps. The SAS Kernel is supported over both Linux and Windows. To evaluate the strength of the SAS framework, third-party softwares for 3D camera from UC, Berkeley and renderer from UC, Davis are used. No source code modification in these third party softwares was needed and these end-devices could easily interface with SAS Kernel by only specifing device configuration in simple XML file.

We compare the performance of SAS Kernel in terms of overheads incurred on a) End-to-End Delay, b) CPU, and c) Bandwidth. We perform two experiments: Experiment 1) SAS Kernel consists of one gateway with 1 to 6 bundles with 2 video streams each, Experiment 2) SAS Kernel consists of two gateways, each with 1 to 6 bundles with 4 video streams each, and 2 output devices (renderer and the other gateway). So, each gateway receives total 24 instreams and sends 24 corresponsing outstreams. It is to be noted that 12 and 24 streams per site is a large workload in DIME scenario in terms of bandwidth (120 to 240 Mbps), CPU (24 to 48 threads), and current applications. Thus, the evaluation highly stresses the system. Moreover, multiple gateways can be spawned to balance the load in the event of dramatic increase in streams. For repeatability, we use a recorded creative dance performance. For the gateway server, we use 4 Dell Precision 670 with dual Intel Xeon processor.

End-to-End Delay Overhead: The major goal of the kernel is to support real-time streaming even under heavy loads. We evaluate the total delay overhead added by the SAS Kernel wherein total delay is the difference between the entry time of a frame and the exit time of that frame from the SAS Kernel. Figures 8(a), 8(b) show that the total delay is less than 3 milliseconds even for 12 concurrent streams, and increases minimally on using 2 sites and 24 streams. This shows that SAS Kernel meets the soft real-time requirements of streaming while efficiently providing SAS.

CPU Overhead: It is important for SAS Kernel to scale in terms of CPU demands as large number of end-devices is added to the system. As shown in Figures 8(c), 8(d), the average CPU overhead ranges between 2% for 2 streams to 20% for 12 streams in Experiment 1. On doubling the number of sites and streams, the average



(a) SAS Kernel delay over single gateway (b) SAS Kernel delay over multi-gateways (c) SAS Kernel CPU demand over single gateway (d) SAS Kernel CPU demand over multi-gateways

Figure 8: SAS Kernel Evaluation

CPU requirement only increases to 10% for 4 streams and 30% for 24 streams. This emphasizes that SAS Kernel demands low CPU even when large number of sensors are connected to it.

Bandwidth Overhead: SAS Kernel adds S-RTP header on the data packets and uses Google Protobufs for marshalling S-RTP frames. Some DIME applications are bandwidth hungry, so it is important that SAS Kernel itself does not add too much bandwidth overhead. For the current implementation of the SAS Kernel, only a fixed cost of 22 bytes per frame is incurred as S-RTP header. The Google Protobuf only adds 4 bytes to the header. Thus, a total of 26 extra bytes per frame over frame size ranging from 2KB to 30KB for 3D-video frames and 140Bytes of audio frames is incurred.

7. RELATED WORK

SAS Kernel synthesizes ideas from service gateways, network services, and operating systems. We discuss the related work in these research areas. In [9], [10], [13], [14] general architectures for home, sensor, and streaming gateways, supporting small set of functionalities like protocol translation, media transcoding, admission control, data processing, synchronization are presented. These gateways focus on a small subset of the SAS requirements and hence fail to provide major SAS services.

In network services, OSGi [7] is a java-based service platform for home networks allowing service providers to dynamically load and deliver services to the end users. However, it is very cumbersome to build complex systems like multi-correlated streaming over low-level OSGi [3]. In [12], user-configuration is used for setting class of service policies in routers. Our approach however, focuses on using user configurations to manage multi-streaming sessions.

SAS Kernel also draws concepts from operating systems like application-level functions in [4] and interface for run-time functions in [18]. In [2], resource containers are presented to provide resource management over processes and threads for network servers. Compared to [2], our approach is at a higher level of abstraction spanning across sites, sessions and streams.

8. CONCLUSION

Our main thesis is that multi-streaming in DIMEs should be modeled as a real-time, generic, flexible, scalable, and robust Streaming as a Service (SAS) for highly correlated sensory data over the Internet. We introduce the concept of SAS and its implementation in the distributed SAS Kernel. SAS Kernel is a proof-of-concept architecture of this SAS model. SAS Kernel supports various types of sensors, transport and session protocols, as well as dynamically loaded functions such as congestion control, compression, and synchronization. Our experiments in a real DIME testbed indicate that SAS Kernel is successful at providing the service without much overhead time-wise (i.e., delay) and space-wise (i.e., bandwidth).

9. ACKNOWLEDGMENT

This research is supported by grants NSF CNS 09-64081, NSF CNS 08-34480, NSF CNS 07-20702, and NSF CNS 10-12194. The presented views are those of authors only.

10. REFERENCES

[1] P. Agarwal et al. Bundle of streams: Concept and evaluation in distributed interactive multimedia environments. In *ISM*, 2010.
[2] G. Banga et al. Resource containers: a new facility for resource management in server systems. In *OSDI*, 1999.
[3] H. Cervantes et al. Beanome: A component model for the osgi framework. In *Software infrastructures for component-based applications on consumer devices*, 2002.
[4] D. Engler et al. Exokernel: an operating system architecture for application-level resource management. In *SOSP*, 1995.
[5] G. Kurrilo et al. Immersive 3d environment for remote collaboration and training of physical activities. In *VR*, 2008.
[6] B. Lampson et al. Reflections on an operating system design. *Commun. ACM*, 19:251–265, 1976.
[7] D. Marples et al. The open services gateway initiative: an introductory overview. *IEEE Commun.*, 39(12):110 –114, 2001.
[8] K. Nahrstedt et al. Next generation session management for 3d teleimmersive interactive environments. *MTAP*, 51:593–623, 2011.
[9] S. Roy et al. A system architecture for managing mobile streaming media services. *Distributed Computing Systems*, 0:408, 2003.
[10] P. Schramm et al. A service gateway for networked sensor systems. *Pervasive Computing*, 3(1):66 – 74, 2004.
[11] R. Sheppard et al. Advancing interactive collaborative mediums through tele-immersive dance (ted): a symbiotic creativity and design environment for art and computer science. In *ACM Multimedia*, 2008.
[12] Y. E. Sung et al. Modeling and understanding end-to-end class of service policies in operational networks. In *SIGCOMM*, 2009.
[13] D. Valtchev et al. Service gateway architecture for a smart home. *IEEE Commun.*, 40(4):126 –132, 2002.
[14] M. Weihs. Design issues for multimedia streaming gateways. *Mobile Communications and Learning Technologies*, 0:101, 2006.
[15] W. Wu et al. Implementing a distributed tele-immersive system. In *ISM*, 2008.
[16] W. Wu et al. "i'm the jedi!" - a case study of user experience in 3d tele-immersive gaming. *ISM*, 2010.
[17] C. K. Yeo et al. A survey of application level multicast techniques. *Computer Commun.*, 27(15):1547–1568, 2004.
[18] G. Zhenyu et al. R2: An application-level kernel for record and replay. In *OSDI*, 2008.