Bellini: Ferrying Application Traffic Flows through Geo-distributed Datacenters in the Cloud

Zimu Liu[1], Yuan Feng[2], and Baochun Li[1]

[1]Department of Electrical and Computer Engineering, University of Toronto
[2]Department of Computing, Hong Kong Polytechnic University

***Abstract*—Thanks to the "on-demand" nature of cloud computing, a large number of applications have been recently migrated to the cloud. To take full advantage of superior connectivities between geo-distributed datacenters, application traffic can be "ferried" through the cloud to provide better service and user experience. However, implementing and deploying such inter-datacenter protocols for various applications, such as messaging, streaming and conferencing, are not without challenges, due to complex requirements of applications and unique characteristics of datacenters. In order to simplify the design and implementation of new inter-datacenter protocols, we design and implement a new system framework, called *Bellini*, in this paper. *Bellini* provides customizable elements shared by the new category of inter-datacenter protocols, including a variety of transport protocols, routing policies, and rate allocation strategies. *Bellini* is also optimized to perform well in virtual machines, utilizing available resources efficiently. With case studies on video conferencing and messaging, we demonstrate the benefits of *Bellini* when it comes to designing and evaluating new inter-datacenter protocols to serve the needs of cloud-based applications.**

I. Introduction

The proliferation of cloud computing using geographically dispersed datacenters has clearly demonstrated its advantages in cost reduction and scalability. Moving traditional applications to the cloud allows the convenience of "pay-as-you-go" when it comes to using resources on demand, enabling these applications to scale up naturally to meet surging user demand. With applications deployed in datacenters, it is common to see that application traffic flows are relayed by satellite datacenters to backbone datacenters. Chen *et al.* showed that inter-datacenter traffic now accounts for a significant amount of the total traffic through the datacenter egress router [1].

On the other hand, with the ubiquitous use of smart mobile devices for streaming and interactive conferencing sessions, geo-distributed datacenters in the cloud can be used to offer *better performance* to a wide variety of mobile applications. Empirical studies reveal that links between geo-distributed datacenters often offer higher capacities than peer-to-peer connections between two end hosts over the public Internet [2]. Therefore, it is conceivable to design new application-layer protocols that take advantage of higher link capacities in inter-datacenter networks, operated by cloud service providers.

The objective of these new *inter-datacenter* protocols is to "ferry" application traffic flows via a collection of datacenter-to-datacenter paths in the inter-datacenter network, while each of these paths may involve multiple hops. With these new protocols, packets in multiple streaming or conferencing sessions can be routed through a high-capacity inter-datacenter network, as if they are traveling around the world in chartered private flights with minimal congestion, rather than cruise ships with long lines waiting for embarkation.

Yet, research is still in its infancy when it comes to how new inter-datacenter protocols can be designed to serve the needs of a variety of streaming, conferencing, and messaging applications, with many research questions remaining open. A major roadblock is that such research requires real-world implementations of new protocol designs, which can be readily deployed in actual datacenters. Such implementations are complex and time-consuming to be designed and realized.

In this paper, we present a new system framework, called *Bellini*, that identifies and incorporates common system elements that are needed by prototype implementations of a class of inter-datacenter protocols. *Bellini* makes it much more convenient to develop, test, and evaluate new protocols in a realistic cloud platform with geo-distributed datacenters, such as Amazon EC2. From the ground up, *Bellini* is designed to efficiently utilize resources in VMs: it uses asynchronous I/O to process incoming and outgoing packets, with high performance in terms of packet processing rates. In order to offer better flexibility so that it can be used by different inter-datacenter protocols, *Bellini* supports one-to-one, one-to-many, and many-to-many communication, and traffic can be split into multiple paths, each transmitted over multiple relays. Design patterns are also widely used, making it convenient to plug in customized protocols and components.

We evaluate the flexibility and performance of *Bellini* through two case studies: video messaging and multi-party conferencing. Our case studies span a range of protocols for routing application traffic flows, including choices of transport protocols, routing policies, and flow assignment strategies. We show how different design choices can be plugged into *Bellini* and tested in datacenters with ease. Our experimental results, obtained by running *Bellini* instances in actual datacenters in the Amazon EC2 cloud, demonstrate the performance we are able to achieve with the framework.

The remainder of this paper is organized as follows. In Sec. II, we discuss the design objectives of the *Bellini* system framework. In Sec. III, we present the design and implementation in detail. In Sec. IV, we study two cases and show how *Bellini* is used to facilitate their development and evaluation.

We conclude the paper with a discussion of related work (Sec. V) and concluding remarks (Sec. VI).

II. Design Objectives

Intuitively, streaming, messaging, and conferencing applications are typically bandwidth-demanding and delay sensitive, and they may benefit most from high-capacity inter-datacenter networks. Since *Bellini* is designed to facilitate the rapid prototyping of inter-datacenter protocols, we wish to identify shared elements of these applications and implement them in a system framework. Let's first examine three representative class of applications:

- ▷ *On-demand streaming applications.* In these applications, a video stream is typically transmitted in an application-layer multicast session, from a media source server to a group of subscribing users.
- ▷ *Messaging applications.* A media message, such as a video/audio clip, is to be shared from one user to another. Though these applications are best-effort in nature and are not as sensitive on delays as streaming, messages will need to be received with a reasonably slow delay, since users may interact in these applications.
- ▷ *Multi-party video conferencing applications.* A group of users, each serves as a video source, are involved in a conferencing session and transmits her/his video stream to other participants. Due to the all-to-all broadcasting nature of these conferencing sessions, they can be potentially bandwidth-demanding and delay-sensitive.

Considering a variety of requirements on application performance, different inter-datacenter protocols will need to be designed to cater to the needs of different applications. Despite their different requirements, application traffic will be sent via the inter-datacenter network, with solutions to two problems. At a high level, they need to manage datacenter nodes so that they can be used to "ferry" traffic flows belonging to a number of concurrent sessions, each involving a group of participants. At a low level, packets are to be transmitted from their sources to the inter-datacenter network, and then relayed to their final destinations, possibly via a number of datacenter-to-datacenter paths, with multiple hops along each path.

Bellini is designed to provide solutions to these problems, as they constitute the shared elements in inter-datacenter protocols. To get started, we first look at the traffic patterns: the traffic can be as simple as a source-destination pair involving two datacenters, or as complex as involving multiple sources and destinations. To organize these traffic flows *Bellini* adopts a two-tier hierarchy, as shown in Fig. 1: A *conference* consists of several nodes participating in the same event, such as the sharing of a media message; a conference further contains one or multiple concurrent *sessions*, each of which has a specific source and a set of destinations. Taking a multi-party video conference as an example, all participants form a conference, and video broadcasting streams originated from each participant are considered as sessions.

Once a session has been established, how should packets be transmitted? Governed by an inter-datacenter protocol, packets

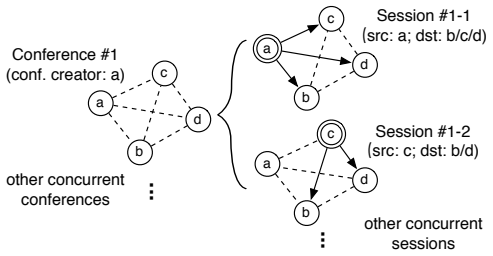


Fig. 1. Organizing traffic flows in *Bellini* with a two-tier hierarchy.

could be sent to one or multiple destination datacenters, using reliable or unreliable connections, through direct paths, multi-hop relay paths, or multicast paths, depending on different application scenarios. For some delay-sensitive applications, it is also possible that a packet is copied and transmitted through different paths to the same destination. In order to embrace a wide variety of routing policies, including *unicast*, *multicast*, *multi-hop* transmission, and *multi-path* transmission, *Bellini* is designed to provide a flexible packet router at the application layer, allowing traffic flows to be sent to an arbitrary node and forwarded by any intermediate nodes. Furthermore, in order to fully utilize inter-datacenter capacities, it is also common to see that rates of different traffic flows are dynamically adjusted. With different application requirements considered, it is critical to support both flexible routing policies and adjustable flow assignment, such that *Bellini* can easily support a wide variety of applications with different requirements and objectives.

We now elaborate on two main objectives as our system framework is being designed: *flexibility* and *performance.*

Flexibility. Designed as a system framework that supports a variety of cloud-based applications, *Bellini* should be highly flexible, so that it helps to simplify inter-datacenter protocol prototyping for networking researchers. To be specific, *Bellini* should allow inter-datacenter protocols to customize their own routing policies and rate assignment strategies, based on certain requirements for different applications. To facilitate such customization, an easy-to-use interface must be provided for the applications to activate available components and configure parameters. To achieve this goal, *Bellini* supports both *API-based control* and *file-based configuration*.

With respect to the API-based control, *Bellini* provides a set of programming interfaces for applications to monitor and control every detail of the underlying transmissions in an online fashion, including decisions on which path should packets be sent through, or which alternative path should be used if congestion is detected. In practice, *Bellini* is controlled through three abstract classes: `IConference`, `IRouting` and `IFlowAssignment`. By deriving C++ classes from these abstract classes, a new inter-datacenter protocol can easily create conferences, and change routing policies and flow assignment strategies. Taking customized routing in Fig. 2 as an example, the `MyRouting` class derived from `IRouting` instructs *Bellini* to periodically load application-specific routing decisions for each session. When necessary, the application can also reload routing policies of a given session.

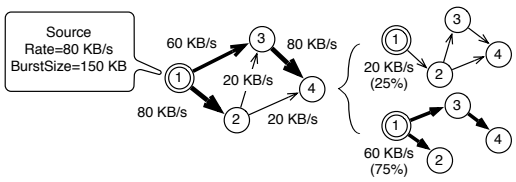
```
class MyRouting : protected IRouting {
    /* app-specific variables and functions */
};

RoutingList MyRouting::routingDecisions(const ConfIDType confId,
const SessionIDType sessionId) { // declared in IRouting
    RoutingList rlist;
    /* app-specific routing algorithms */
    rlist.addRoutingEntry(/* a routing policy */);
    // ...
    return rlist;
}

{ /* initialization */
    MyRouting myRouting(/* init. values */);
    gBellini.registerRoutingObj(myRouting);
    // ...
}

{ /* when certain event occurs */
    /* make some adjustments */
    myRouting.adjustSomething(/* ... */);
    /* force Bellini to call IRouting::routingDecisions and
       reload routing polices */
    gBellini.reloadSessionPolicies(confID, sessionId);
}
```

Fig. 2. Implementation of customized routing algorithms using the `IRouting` programming interface.

In addition to API-based control, *Bellini* also supports the JSON-based configuration file. By writing a human-readable lightweight JSON file, *Bellini* allows users to config a wide variety of settings, such as routing policies and rate assignment strategies for individual conferences and sessions, without writing any C++ code. Fig. 3(a) lists a segment of a *Bellini* configuration file, showing a transmission session from the source node 1 in conference 1. We can see that there exist two routing policies from the source (node 1) to destinations (node 2 and 4), with their respective flow rate assignments. Following such a mix of multi-hop multi-path policies, packets will be transmitted, forwarded, and duplicated through designated paths, as shown in Fig. 3(b).

```
// ...
"ConferenceList": [
{
  "ConferenceID": 1,
  "StartTime": "2012-09-02 14:22:31 UTC",
  "Participants": [1,2,3,4],
  "SessionList": [
  {
    "SessionID": 1,
    "Protocol": TCP,
    "SrcNode": 1, /* Source Node ID */
    "DstNodes": [2,4], /* Destinations */
    "AverageRate": 80, /* 80 KB/s */
    "MaxBurstSize": 150 /* 150 KB at burst */
    "RoutingList": [
      { "FlowWeight": 25, "PathList": ["2,4","2,3,4"] },
      { "FlowWeight": 75, "PathList": ["2","3,4"] },
    //...
```

(a) A segment of the *Bellini* configuration file.



(b) The corresponding traffic flows.

Fig. 3. An example of configuring *Bellini* for multi-hop multi-path transmission session from node 1 to node 2 and 4.

Performance. Since *Bellini* is designed to support the implementation of real-world applications to be deployed in actual cloud datacenters, it should be implemented with performance and scalability in mind. In particular, it is critical for *Bellini* to achieve the best possible performance, such that all resources purchased in datacenter are utilized efficiently. Furthermore, when the system scales up, *e.g.*, multiple nodes are forming multiple concurrent transmission sessions, which is the norm in reality (*e.g.*, in the video streaming scenario), *Bellini* should be capable of handling a large number of concurrent sessions, each maintaining a high transmission rate.

In order to achieve this goal, the design of *Bellini* is guided by the asynchronous event-driven paradigm, in which incoming events (*e.g.*, packet reception) are processed by corresponding handlers, and subsequent events may be generated for further processing. The advantage of the event-driven paradigm is two-fold. *First*, the event-driven engine incurs less CPU and memory overhead. When the workload concurrency of a cloud node is high, the event-driven model only uses a fixed number of threads, while the "thread-per-connection" model will create a large amount of working threads, leading to excessive overhead of thread context switching. *Second*, with the help of the event-driven engine, components inside *Bellini* can be loosely coupled. As long as event handlers from different components are appropriately registered for specific events, event-based workflows are naturally formed and then components can work seamlessly.

III. Implementation

With our design objectives in mind, we have designed and implemented *Bellini* from scratch, using C++ and the Boost `asio` asynchronous I/O library. To realize new cloud-based applications, inter-datacenter protocols can be easily built on the *Bellini* framework, and then deployed in VMs at geo-distributed datacenters. Using the interface provided by *Bellini*, application instances (henceforth called *nodes*) can establish or join one or more conferences and corresponding transmission sessions. Within a conference, *Bellini* assists applications to send and receive data to/from instances running on other VMs. For different application traffic, combinations of routing policies and rate control strategies can be specified and executed. In order to thoroughly monitor the performance of the entire system, *instrumentation* units are embedded in *Bellini*. We now present further details on the design and implementation of the *Bellini* framework.

Conference and Session Management

Bellini manages all active transmissions using the aforementioned conference-session hierarchy. When a conference is initiated by the application, using either a configuration file or the `IConference` programming interface, *Bellini* node will create an object that corresponds to this conference, and register it in a conference listing server. With the conference created, the application can further establish multiple concurrent sessions for actual transmissions among participating nodes. Within a session, data from the session source will

be sent to designated destinations for storage or playback. In order to maximize its flexibility, *Bellini* allows the application to customize each individual session, by specifying its routing policies and flow assignment strategies in the interface.

If a node in the conference serves as the source of a session, *Bellini* retrieves content from the application running on this node, and then supply data packets to underlying transmission components. In reality, content may be files (*e.g.*, digital photographs and asynchronous messages) or streams (*e.g.*, conferencing videos and on-demand videos). Furthermore, streams can be further categorized into two groups: adaptive bitrate streams, which detects the available bandwidth and adjusts the stream quality accordingly, and non-adaptive streams. Since *Bellini* is designed to support different applications, we accommodate all three types of content, by properly converting them into a flow of data packets to be transmitted. In addition to using actual content from applications, *Bellini* also provides a flexible data generator that can emulate different types of data sources, which facilitates the testing of new inter-datacenter protocols.

Flexible Source Routing

When designing routing support in *Bellini*, an important design choice is whether the routing decision should be made by the source node of a session in a centralized fashion, or by individual nodes in the session in a distributed manner. Different from traditional peer-to-peer routing protocols, *Bellini* is designed to transmit traffic flows through datacenters. Since the total number of datacenters is relatively small, only a limited number of nodes are involved in inter-datacenter protocols. In that sense, the complexity of centralized routing algorithms can be reduced in the cloud. Furthermore, thanks to the excellent connectivity and abundant bandwidth between datacenter VMs, running a centralized algorithm becomes much more convenient. Taking full advantage of the cloud infrastructure, *Bellini* adopts the source routing paradigm to achieve better performance.

With source routing chosen, we have design flexible structures in *Bellini* to describe different routing policies, including direct, multicast, multi-hop, and multi-path transmissions. As shown in Fig. 4, the `RoutingInfo` to be embedded in the packet header consists of one or more segments of `Pathlet` data, with each `Pathlet` segment representing a next-hop node. Within a `Pathlet` segment, the next-hop node is identified by a unique node identifier (Node ID), and the data in `RoutingInfo` indicate how this next-hop node should handle the packet. Since several destinations may exist in a session, we reserve the highest bit in the Node ID to identify whether the next-hop is one of the destinations. With the help of these two data structures, we can easily specify different routing policies in *Bellini*. Table I lists a few examples of policies and corresponding representations in *Bellini*.

Adjustable Flow Assignment

In addition to source routing, another important feature of *Bellini* is the support of customizable flow assignment, where

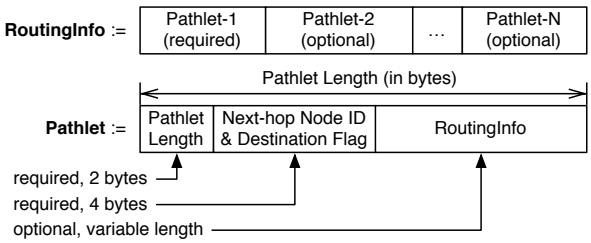
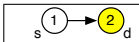
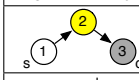
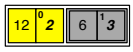
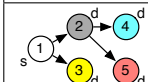
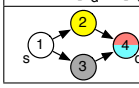
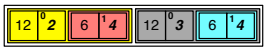


Fig. 4. The `RoutingInfo` and `Pathlet` structures for source routing.

TABLE I
EXAMPLES OF REPRESENTATIVE ROUTING POLICIES REPRESENTED BY THE `RoutingInfo` DATA STRUCTURE.

Routing	Category	Representation
	Direct	
	Multi-hop	
	Multicast	
	Multi-path	

Note: "s" indicates a source; "d" indicates a destination; the numbers in boxes represent values in fields of `RoutingInfo` and `Pathlet` structures; the superscript number in the Node ID field indicates the highest bit.

data packets from a session can be split into several flows, and each flow is transmitted at a given rate following a specific routing policy. As discussed previously, *Bellini* executes routing decisions at the granularity of individual packets in the source node, by embedding the routing information in each packet. Such a flexible design allows us to send packets carrying different `RoutingInfo` headers, each of which represents the path of a traffic flow, and then packets will be naturally split into multiple flows by the dispatcher.

In order to implement flow assignment in each session, we organize all active flows in the same session as a traffic flow list at the source node, and each entry in the list represents a traffic flow and has the corresponding `RoutingInfo` header cached. When a packet in a session to be sent, *Bellini* randomly choose an entry from this session's flow list, and embed the cached routing information into this packet. By allowing the application to assign customized statistical distribution to entries, packets will be sent in different flows at given probabilities, and thus different portions of traffic flows will be transmitted in a controlled manner. To further enhance the flexibility of *Bellini*, an application is able to dynamically change the flow assignment distribution of each individual session, by inheriting the `IFlowAssignment` abstract class.

Supporting Concurrent Conferences and Sessions

Bellini is designed to support cloud-based applications, and it is the norm to have multiple ongoing conferences and sessions, when the system scales up. Furthermore, in a large-scale system, conferences and sessions may start and end frequently. Hence, it is critically important to efficiently

support concurrent traffic sessions in *Bellini*'s implementation.

First, the *Bellini* dispatcher is implemented with conference/session awareness. For each session, its corresponding routing and flow assignment decisions will be loaded into a per-session internal object for speedy access. In this way, each session can operate separately without affecting each other. *Second*, we must think carefully how outgoing packets of the dispatcher — particularly, packets from different sessions but to the same next-hop node — should be transmitted, to support a large number of concurrent sessions. Intuitively, a solution is to establish separate connections from the local node to the next-hop node, with each connection used by one ongoing session. However, such an approach will incur additional overhead (*e.g.*, handshake and slow-start) when creating new connection for each session. To ensure system performance, multiplexed node-to-node connections is adopted in *Bellini*, by sending packets designated to the same next-hop node through the same connection.

In order to implement such an efficient and scalable design, we create a data feeder object for each next-hop node, and each data feeder collects packets from the *Bellini* dispatcher to be sent to a particular next-hop node. As shown in Fig. 5, within each data feeder, we maintain a list of per-conference queues to buffer packets from different conferences, rather than queueing all packets in a shared buffer. On one hand, such a design provides the flexibility to balance or prioritize ongoing conferences in *Bellini*. As a transmission opportunity comes, the data feeder can pop a packet from per-conference queues in a round-robin manner or in a prioritized order specified by the application. On the other hand, once a conference is terminated, associated queues can be immediately removed to eliminate useless packets, instead of scanning shared queues. It is also worth noting that as multi-hop transmissions are allowed, queues in an intermediate node may overflow when the congestion occurs. In such a case, *Bellini* will notify all involved nodes so that the application can take proper actions.

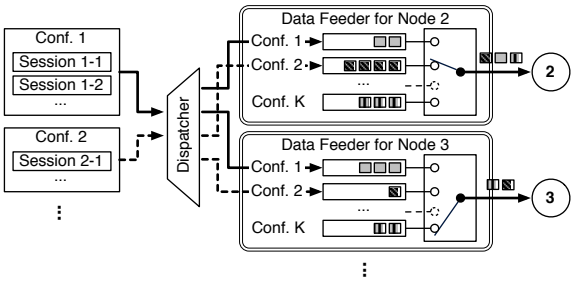


Fig. 5. An illustration of data feeders in *Bellini*.

Implementing High-Performance Packet Delivery

With packets queued in data feeders, we are ready to deliver them to the corresponding next-hop nodes. In out implementation, *Bellini* provides high-performance packet delivery to other running *Bellini* nodes. Applications can choose different transport protocols in *Bellini*, with support for both TCP and UDP as transport protocols. In order to achieve a reasonable fairness when UDP and TCP compete for bandwidth, TCP-friendly rate control (TFRC) is activated in each UDP connection as the flow control algorithm.

With multiplexed transmission, each TCP/UDP connection to a remote node will be associated with a corresponding data feeder. Once a transmission opportunity occurs, *Bellini* immediately requests a packet from the associated data feeder and send it out. Meanwhile, *Bellini* asynchronously waits for any incoming packets from the network. Upon receiving a packet, a reception completion handler will be triggered and the packet will be sent to the dispatcher for further processing. In addition to data packets, all control packets, such as session management messages or congestion notifications, are sent over dedicated connections, to timely manage the *Bellini* system.

Network Coding

In recent years, we have observed that *random network coding* has been widely used for peer-to-peer transmissions. With satisfactory performance reported in the literature [3], *Bellini* supports random network coding, so that researchers can simply activate network coding as a component, and then evaluate its suitability in new inter-datacenter protocols.

As network coding is a rateless erasure code, we decide to incorporate it with UDP and TFRC in *Bellini*, to provide an alternative transmission protocol supporting both error control and flow control. To be specific, packets to be sent are linearly combined to produce coded packets, using random coefficients in $GF(2^8)$. At a receiving node, the decoding is performed progressively using the Gauss-Jordan elimination. Any relay node may also produce coded packets by performing similar operation on the received coded packets. To provide the best possible performance when the network coding engine is enabled, we have included a fully optimized network coding codec in *Bellini*. Our accelerated codec is able to conduct network coding in a parallel manner using SIMD instructions.

Instrumentation and Supporting Scripts

Since *Bellini* is primarily designed as a real-world deployment platform to evaluate new inter-datacenter protocols, the ability to evaluate its runtime performance is a must. We have implemented instrumentation units in *Bellini* to closely monitor various performance metrics, *e.g.*, the TCP/UDP throughput, per-session packet forwarding rates, and end-to-end delays. Every 30 seconds, these performance metrics are reported to the `gStatistics` object provided in the interface, so that the application is able to monitor the *Bellini* runtime. For the convenience of offline performance analysis, periodic performance reports are also written to logs.

Last but not the least, *Bellini* includes an extensive set of deployment scripts that provides "turn-key" solutions when it comes to deploying multiple executable instances and configuration files to their respective datacenters, launching them for execution, and collecting logs after they are terminated. Batch processing scripts have also been provided to automate the deployment of a large number of performance tests (perhaps with

different configuration settings) without human intervention.

IV. Case Studies

We now use two case studies to show how *Bellini* can facilitate the implementation and evaluation of new inter-datacenter protocols. In our case studies, the flexibility and performance of *Bellini* are assessed thoroughly.

A. Video Messaging with Minimized Traffic Costs

With increasing uses of messaging applications on mobile devices, it is conceivable that, not only text, short videos can also be messaged. And, such messaging is not as sensitive to end-to-end delays as streaming and conferencing applications. If videos to be messaged are transmitted over inter-datacenter networks with high-capacity links, we can focus more on the operational costs of running such video messaging service. We discover that percentile-based charging models that are typically used in inter-datacenter networks may provide further opportunities to reduce operational costs: if some traffic is already generated on one link, transmitting less traffic in subsequent time intervals will be a waste of capital investment. Therefore, a possible way to reduce costs is to design routing and flow assignment for traffic flows so that the under-utilized time intervals are minimized as much as possible.

To achieve this goal, a set of algorithms has been designed to minimize the cloud operator's costs on messaging traffic, by optimally routing flows in an online fashion [4]. As *Bellini* is designed to be flexible, using it to build a prototype of such a video messaging application—possibly using intermediate datacenters as relays—is a breeze. By implementing the new algorithms as C++ objects derived from `IRouting` and `IFlowAssignment`, video traffic across inter-datacenter links can be split and transmitted along multiple multi-hop paths. Thanks to such a flexible support of the *Bellini* framework, the prototype of the video messaging service has no more than $1,000$ lines of C++ code.

With such a prototype built on *Bellini*, we use the cloud deployment scripts to deploy instances of our prototype in medium VMs in 7 Amazon EC2 datacenters and evaluate the performance. First, we investigate the relation between the packet processing rate and the resource usage of *Bellini*, by sending an increasing number of randomly generated video messages. After collecting all the logs, we group performance reports by 5-min intervals, and then derive the average packet processing rate and CPU/memory usage within each interval. As shown in Fig. 6, there exists a strong linear correlation between the processing rate and the CPU usage, which makes it possible to estimate *Bellini*'s performance given the processing power of a VM. Such a predictable performance of *Bellini* is mainly attributed to the low CPU overhead incurred by the high-performance event-driven engine. With respect to the memory usage, we observe that the memory usage is reasonably low even at a high processing rate, implying that packets are processed in a timely manner.

We then evaluate the time consumed internally for packet processing, to verify the effectiveness of online routing and flow assignment. Since each packet will be first combined with a routing header to execute the customized routing and flow assignment decisions, we plot the CDF of the processing time consumed on such operations in Fig. 7. It is shown that the average processing time for each packet is as low as 1.89 μs and the standard deviation is no more than 0.1 μs. Recall that all packets, including local packets and incoming packets from other nodes, will be forwarded to corresponding next-hop nodes by the dispatcher. Fig. 8 further examines internal processing times in *Bellini*'s dispatcher. With an average of 5.47 μs/packet observed, we can derive that the total processing time consumed by these two major steps is around 7.4 μs/packet. These observations reveal that *Bellini* is able to achieve excellent performance in real-world settings.

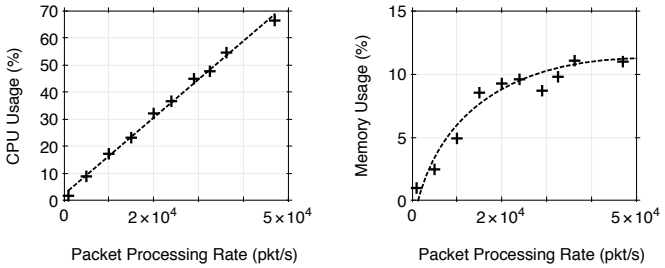


Fig. 6. Comparisons between the packet processing rate and CPU/memory usage of *Bellini*.

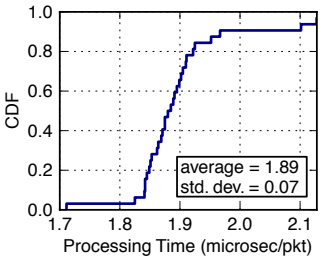


Fig. 7. Processing time consumed for routing and flow assignment.

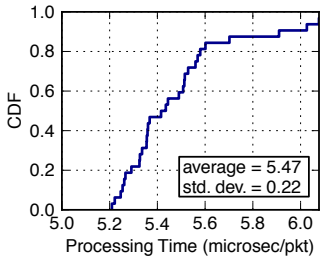
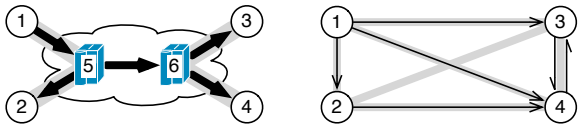


Fig. 8. Processing time consumed on dispatching packets.

B. Video Conferencing using Inter-Datacenter Networks

Traditionally, multi-party video conferencing protocols are designed to use a peer-to-peer architecture. However, due to a lack of bandwidth between nodes over the Internet, its quality is not satisfactory. Since datacenters in the cloud are typically connected via dedicated links, it is conceivable that throughput may be higher by relaying the conferencing session over the inter-datacenter network. In this case study, we would like to use *Bellini* to validate the idea of cloud-based video conferencing, named *Airlift* [2]. By delivering live video conferencing streams via datacenters, *Airlift* algorithms try to use network coding and optimal flow control to maximize the total throughput, without violating end-to-end delay constraints. Since *Bellini* supports network coding assisted multi-hop transmission, it is trivial to setup the *Bellini*: after turning on the network coding engine, *Bellini* is able to automatically



(a) Cloud (b) Peer-to-Peer

Fig. 9. Topologies supported by different *Bellini* configurations, with one of sessions highlighted. Note that nodes in Toronto, Waterloo, Beijing and Shanghai are numbered with 1, 2, 3 and 4, respectively; Virginia and Tokyo datacenters are represented by 5 and 6.

encode packets, forward them through designated paths using the optimal flow assignment computed by *Airlift* algorithms, recode them whenever necessary, and finally decode coded packets at destinations for playback.

To conduct a preliminary test of *Airlift*, we first deploy four *Bellini* instances as conference participants, in PlanetLab nodes located in Toronto, Waterloo, Beijing, and Shanghai, respectively. These four instances forms 4 concurrent sessions in a conference, with each session corresponding to a video source at one of participants. As illustrated in Fig. 9(a), we further deploy two *Bellini* instances in the Amazon EC2 Virginia and Tokyo datacenters as relay nodes for this conference. By analyzing logs produced by instrumentation facilities in *Bellini*, we have observed that the throughput can achieve up to 1.9 Mbps. On the other hand, it is observed that the end-to-end delay, with an average of 191 ms. For comparisons, we reconfigure 4 *Bellini* instances in PlanetLab to execute optimal peer-to-peer based packet delivery (shown in Fig. 9(b)). The experiment shows that the achievable throughput is merely 142 kbps with an end-to-end delay of 157 ms on average.

Encouraged by such observations, we deploy *Bellini* instances to all Amazon EC2 datacenters around the world, as relays for multiple source-destination pairs of PlanetLab nodes. By writing different configuration files and running scripts, all *Bellini* instances are launched automatically to conduct various experiments. Table II summarizes the runtime traces between representative cities in different continents. Overall, the collected logs show that throughput improvements are around 3–24 times.

TABLE II
PERFORMANCE TRACES BETWEEN CITIES IN DIFFERENT CONTINENTS.

Cloud / P2P	Total throughput (Mbps)	End-to-end delay (msec)
Toronto-Beijing	14.11/4.04	169.8/142.3
Vancouver-Berlin	34.32/1.38	137.2/104.9
Seoul-Rio	20.32/2.32	228.6/203.5

V. Related Work

In the literature, very little effort has been devoted to the design and implementation of a flexible and high-performance framework to support inter-datacenter protocols. Before the era of cloud computing, several peer-to-peer frameworks, such as PeerSim [5], were designed to simulate or emulate peer-assisted data transmission protocols. Although these frameworks can be configured to imitate datacenter networks, they cannot help researchers to implement, deploy, and evaluate actual applications in real-world cloud. With respect to application packet forwarder, RON [6] was implemented to route packets in an application-layer overlay network. But, RON only supports simple multi-hop routing, and cannot split a session into multiple flows to conduct fine-granularity control. It also fails to provide a flexible customization interface for cloud-protocol prototyping. To support cloud-based applications, very few frameworks were proposed with focus on data transmission. The SAM framework [7] was proposed solely for the backup traffic in the cloud.

Different from existing work, *Bellini* focuses on inter-datacenter transmission of application traffic flows in the real-world cloud environment. Designed as a flexible system framework, *Bellini* helps researchers to simplify the implementation and evaluation of new protocols that use datacenters in the cloud. With flexibility as the most important design objective, essential elements such as routing and flow assignment, are supported in *Bellini*, and a wide variety of customizations can be achieved through the configuration file or the given programming interfaces with ease.

VI. Concluding Remarks

This paper presents *Bellini*, a system framework that is designed to facilitate the rapid development, cloud deployment, and instrumentation of a wide range of inter-datacenter protocols. Governed by the design objective of flexibility, *Bellini* supports plugging in customer-tailored routing policies and flow assignment strategies, in order to meet the needs of different applications. The implementation of *Bellini* is fine-tuned for performance, so that resources in cloud VMs can be efficiently utilized. *Bellini* supplies most of the features and components that are desired by a variety of applications, such as sharing, conferencing and messaging. Our experiences with two case studies have shown that, *Bellini* makes it feasible to develop and evaluate new protocols using datacenters with ease and satisfactory performance. We will release *Bellini* as an open-source software release so that other researchers may benefit from the framework.

References

[1] Y. Chen, S. Jain, V. Adhikari, Z.-L. Zhang, and K. Xu, "A First Look at Inter-Data Center Traffic Characteristics via Yahoo! Datasets," in *Proc. INFOCOM*, 2011, pp. 1620–1628.

[2] Y. Feng, B. Li, and B. Li, "Airlift: Video Conferencing as a Cloud Service using Inter-Datacenter Networks," in *Proc. IEEE International Conference on Network Protocols (ICNP)*, 2012.

[3] C. Gkantsidis, J. Miller, and P. Rodriguez, "Comprehensive View of a Live Network Coding P2P System," in *Proc. IMC*, 2006, pp. 177–188.

[4] Y. Feng, B. Li, and B. Li, "Jetway: Minimizing Costs on Inter-Datacenter Video Traffic," in *Proc. ACM Multimedia*, 2012.

[5] A. Montresor and M. Jelasity, "PeerSim: A scalable P2P simulator," in *Proc. Intl. Conference on Peer-to-Peer (P2P '09)*, 2009, pp. 99–100.

[6] D. Andersen, H. Balakrishnan, F. Kaashoek, and R. Morris, "Resilient Overlay Networks," in *Proc. SOSP*, 2001, pp. 131–145.

[7] Y. Tan, H. Jiang, D. Feng, L. Tian, Z. Yan, and G. Zhou, "SAM: A Semantic-Aware Multi-tiered Source De-duplication Framework for Cloud Backup," in *Proc. International Conference on Parallel Processing*, 2010, pp. 614–623.