Invenio: Communication Affinity Computation for Low-Latency Microservices

Amit Sheoran
asheoran@alumni.purdue.edu

Sonia Fahmy
Purdue University
fahmy@purdue.edu

Puneet Sharma
Hewlett Packard Labs
puneet.sharma@hpe.com

Navin Modi
modin@alumni.purdue.edu

ABSTRACT

Microservices enable rapid service deployment and scaling. Integrating poorly-understood microservice components into Service Function Chains (SFCs) or graphs limits a provider's control over service delivery latency, however. Orchestration frameworks currently instantiate and place myriads of microservice components without knowing the impact of placement decisions on latency.

In this paper, we explore challenges that service providers encounter in managing complex SFCs, and propose *Invenio* to empower providers to effectively place microservices without prior knowledge of service functionality. *Invenio* correlates user actions with procedure messages in network traces, and computes *procedural affinity* of communication among microservices for each user action. The procedural affinity values can then be used to make placement decisions to meet latency constraints of individual user actions. Our experiments with two microservice-based cellular network implementations demonstrate that placement with *Invenio*-computed affinity values significantly reduces failures by bounding message processing latency, resulting in up to 21% performance gain compared to message count-based placement algorithms, and up to 51% gain over default placement.

CCS CONCEPTS

•**Networks → Network management;** *Data center networks; Mobile networks;*

KEYWORDS

Cloud-native architectures; Microservices; Cellular Networks

ACM Reference format:
Amit Sheoran, Sonia Fahmy, Puneet Sharma, and Navin Modi. 2021. *Invenio*: Communication Affinity Computation for Low-Latency Microservices. In *Proceedings of Symposium on Architectures for Networking and Communications Systems, Lafayette, IN, USA, December 13–16, 2021 (ANCS '21),* 14 pages.
DOI: 10.1145/3493425.3502750

1 INTRODUCTION

Network Functions Virtualization (NFV) has enabled service providers to deploy virtualized instances of Network Functions (NFs) on demand [26]. New service offerings are created by adding one or more NFs to a Service Function Chain (SFC), *i.e.*, a graph of NFs. An existing service can be extended by adding software modules, whereas unpopular features can be removed by deleting modules. Software architectures have evolved to support this rapid pace of deployment, and disaggregated fine-grained microservice designs are now replacing monolithic designs [21, 53].

The power to rapidly add and delete new services comes at a cost, however. SFCs are becoming more complex, and the effort associated with service deployment is growing [29, 51, 61, 62]. Service providers, in an attempt to reduce costs, are increasingly using private or public clouds to deploy services that had traditionally been confined to a single data center and had used carefully-designed proprietary hardware. This *cloudification* poses unique challenges to orchestration frameworks, particularly in instantiating and placing Virtualized Network Functions (VNFs) in an SFC with strict Service Level Agreements (SLAs) [62].

Prior work [45, 48, 49, 55] has shown that network functions (NFs) in systems such as the cellular Evolved Packet Core (EPC) and IP Multimedia Subsystems (IMS) have stringent end-to-end latency requirements and react poorly to unpredictable latency variation. Fortunately, service providers

ANCS '21, Lafayette, IN, USA

DOI: 10.1145/3493425.3502750

can leverage their knowledge of NF functionality and meticulously define SFCs [1, 2] to bound the latency. Virtualization platforms such as Openstack [41], Kubernetes [34], and Docker [16] allow administrators to configure "affinity policies" in NF placement. The affinity policies specify which NFs should be co-located to meet SLA requirements. However, the increasing use of non-standard interfaces and the ongoing integration of 5G core (5GC) [3] into existing 4G network deployments is necessitating extensive manual re-analysis of communication patterns. The diversity of NFs in modern networks and the new 5GC interfaces make determining the SFCs involved in service delivery and the communication affinities between their constituent NFs a time-consuming and error-prone task.

NFs with microservice designs further complicate SFCs. Microservices advocate the use of fine-grained, independent components that can be deployed as autonomous entities communicating via REST-based proprietary interfaces [11, 43]. This results in disaggregation and decomposition of a VNF into multiple smaller VNF Components (VNFCs), and longer, more complex SFCs [21, 62]. Further, lack of standardization in microservice architectures yields VNFCs that play roles that do not accurately map to an NF defined by standards. That is, a VNFC may take the role of several standard-defined NFs and support several network interfaces. Conversely, the functionality of a standard-defined NF may be collectively performed by multiple VNFCs. The ambiguity in the role of VNFCs implies that placement using domain knowledge is insufficient, and we need automated tools to infer communication patterns between microservice components/VNFCs. Our work fulfills this need.

We use information exposed by NFs or their components (VNFCs or microservices) to aid providers [29, 51, 61]. Merely co-locating NFs based on the number of messages they exchange [51], however, can yield unexpected results due to the diversity of workloads. Instead, we propose grouping events triggered by a user action into *procedures*, and computing *procedural affinity* between NFs. A provider can then make placement decisions based on procedural affinity values, together with procedure type distribution and policies. For example, a VNFC used during voice calls, but not for SMS (text-msg), in a cellular network can be placed based on the currently dominant workload type and its requirements. Another example is a drone swarm service [21]. In the drone swarm service, the orchestrator can choose to collocate microservices involved in drone navigation (image capture, image processing, obstacle avoidance, and motion controller) to optimize the placement for drone navigation over routine services such as video upload or logging.

In this paper, we propose *Invenio*, a system for computing communication affinity to aid service providers in deploying control-plane NFs. *Invenio* maps user activity at the network edge to traffic in the network core and computes procedural affinity. *Invenio* is executed after upgrades or policy and service changes. It analyzes a snapshot of traffic to compute affinity values. An orchestrator can then use computed procedural affinity values, in conjunction with current procedure type distribution and policy rules, to make placement decisions. *Invenio* empowers providers to optimize placement to meet SLA objectives, even with upgrades in services and microservices and changing user demands.

In summary, (1) We identify the challenges for a service provider to meet SLAs (§3) and introduce the notion of procedural affinity (§4). (2) We propose *Invenio* for service providers to automatically compute procedural affinity (§5) after isolating messages of a given procedure type for a specific user. *Invenio* can also check NF interoperability and diagnose network problems (§7). (3) We experimentally demonstrate the benefits of placement based on procedural affinity by applying *Invenio* to microservice-based cellular network implementations and measuring the performance of voice-call and text-msg workloads (§6). We find that placement based on *Invenio*-computed affinity results in up to 21% performance gain compared to message count-based placement algorithms, and up to 51% gain over default placement. While our evaluation uses the 4G control plane as a case study, the principles underlying *Invenio* are applicable to the service-based architecture of the 5GC and other microservice-based deployments.

2 MOTIVATION

A control-plane NF can be instantiated on bare metal (as a Physical Network Function (PNF)) or on virtualized hardware (as a VNF), and a VNF can be deployed as a collection of VNFCs. In the rest of this paper, we use the term NF to refer to all three types of instantiations (PNF/VNF/VNFC), and we use the terms VNFC and microservice interchangeably.

The increasing use of clouds to reduce operational costs has yielded scenarios where NFs in an SFC are deployed across multiple physical machines in one or more data centers. Consider Fig. 1 which shows an example microservice-based cellular network for Voice over LTE (VoLTE) that includes wireless access, session management, voice-call signaling, policy control (QoS), and billing. Latency-sensitive NFs (such as signaling and policy) may be connected by high and unpredictable latency links. An orchestrator that cannot instantiate the entire SFC in Fig. 1 on a single machine or rack can identify the NFs exchanging a large number of messages and place them in close proximity. Modern networks offer many services, however, and NFs exchange different types and numbers of messages to support each service.

Not all services have equal impact on user-perceived latency and QoE. For example, interactive services such as

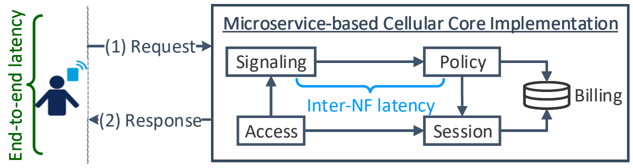


Figure 1: A microservice-based cellular network

voice calls impact user QoE more than non-interactive services such as text-msg or presence services [54]. Orchestrators must reduce the end-to-end latency of interactive services by minimizing the inter-NF latency for NFs handling these services. Simple techniques such as counting total messages exchanged between NFs [51] are not always effective in making placement decisions as they do not explicitly consider the impact of inter-NF latency on user QoE.

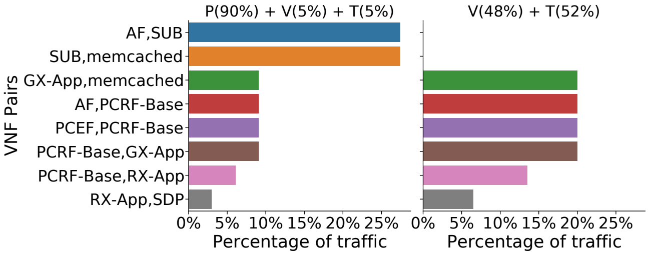


Figure 2: Impact of procedure type distribution (Presence (P), Voice (V), and Text (T)) on number of messages exchanged between NF pairs in a microservice-based VoLTE implementation

Fig. 2 shows the percentage of traffic exchanged by NF pairs (in our implementation in §6.2) for two different procedure type distributions of voice-call, text-msg, and presence services. The plot on the left uses traffic proportions from typical busy-hour IMS traffic [7] in which presence is triggered ~9x more frequently than voice. Clearly, exchanged messages depend on the incoming procedure type distribution and therefore merely using the number of messages for placement [51] may optimize non-interactive services such as presence and degrade user QoE. To meet SLAs for latency-sensitive services, service providers may (a) create dedicated NFs to optimize specific functionality [48], or (b) decompose existing monolithic applications into lightweight microservice components, that are then aggregated by functionality to create NF bundles, and placed together with a higher probability [55]. Manually identifying and configuring bundles can be difficult and error-prone, however.

Our work empowers service providers to easily and automatically react to upgrades and changing user QoE demands. Based on prior research [45, 48, 49, 55], we observe that: (a) NFs typically exchange several messages to complete a seemingly simple user action such as turning on User Equipment (UE) or making a voice call, and (b) Network endpoints only perceive latency in the actions they trigger (*i.e.*, end-to-end latency in Fig. 1) and are oblivious to message exchanges and inter-NF latency within an SFC. User QoE therefore only depends on user action/network response pairs, such as initiating a voice call (action) and hearing a dial tone (network response), or turning on an Internet connection (action) and being connected to a packet access network such as LTE (network response).

3 CHALLENGES

The goal of *Invenio* is to facilitate NF placement by leveraging readily available knowledge of *endpoint actions*. We group events or messages triggered due to a single user action into *procedures*. We then use this procedure information to compute *procedural affinity* (§4) between NFs for each procedure type. The procedural affinity information is used for NF placement. Since NFs in modern networks exchange numerous messages, manually determining control messages that are triggered due to a specific endpoint (or associated user or subscriber) action can be tedious and error-prone. We propose to (a) automatically isolate SFC control messages related to a user, and (b) map each message to an action invoked by that user. We describe the challenges in accomplishing these tasks in the remainder of this section.

Scale and complexity: We need to understand the protocols and messages exchanged by each NF. For example, consider a cellular network EPC (including NFs to interwork with previous generation networks (2G, 3G) and WiFi). Such a deployment can involve 60+ NFs communicating via 15+ protocols over 150+ interfaces using 500+ message types [2, 15]. While many of these NFs are logical, the sheer number of NFs, supported protocols, and message types makes understanding control-plane traffic difficult.

User and session identification: Networks such as cellular networks check user (subscriber) identifiers located in control messages to determine the user associated with a device or a network endpoint, and NFs use these identifiers to enforce policies and bill users. A user is identified by: (a) **Subscriber-ID:** the key used by the network to authenticate a device, identify packets associated with it, and bill the user, and (b) **Session-ID:** the key allocated by an NF to group together messages triggered by a device. Unlike the subscriber-ID, the value of session-ID is not pre-allocated, *i.e.,* NFs allocate a value at run-time. Different protocols and interfaces use different terms to refer to the subscriber-ID and session-ID carrying headers. Table 1 lists example protocol headers used by cellular network protocols.

Since the session-ID is dynamically allocated, the relation between session-ID and subscriber-ID may vary based on the NFs involved in message processing. When a single device creates multiple connections at the same time (such as

Table 1: Example user and session headers

Protocol	Interface Name	Subscriber-ID Header Name	Session-ID Header Name
SIP	Gm [1]	To, From	Call-ID
Diameter	Cx [6]	User-Name, Public-Identity	Session-Id
	Gx [4]	Subscription-Id	Session-Id
S1AP [2]	S1-MME	IMSI, TMSI	eNB-UE-S1AP-ID
HTTP/2 (5G) [3]	N7/N11	SUPI, SUCI	pduSessionId
	Rx/N5	SUPI, SUCI	appSessionId

in EPC), multiple session-IDs may be allocated to the same subscriber-ID. Additionally, an EPC/IMS may create a mapping between the session-ID and the subscriber-ID, and then use the two values interchangeably. Fig. 4 depicts an example where the IMS network uses the User-Name in the *From* header of the SIP protocol [50] to determine the subscriber-ID while interacting with the user, but further messages generated due to this user interaction use other protocol headers, such as the *Public-Identity/Subscription-Id* [19] used when communicating with the Policy and Charging Rules Function (PCRF). After receiving the initial message ((2) CCR in Fig. 4a) from the Policy Charging and Enforcement Function (PCEF), the PCRF creates a mapping between Subscription-Id and session-ID. This mapping identifies the user in all future message exchanges between the PCRF and PCEF (9, 10, 15 and 16 in Fig. 4b omit the subscriber-ID headers and only carry the session-ID header). Therefore, identifying all messages that are triggered due to a user action requires understanding the mapping between session-ID and subscriber-ID.

As in the 4G core, 5GC NFs [3] use headers such as Subscription Permanent Identifier (SUPI) and Subscription Concealed Identifier (SUCI) to identify, authorize, and bill traffic. A user can create multiple sessions with 5GC data networks and therefore the 5GC NFs use session headers such as the pudSessionId in conjunction with the user ID to uniquely identify user sessions. Example headers used in 5CG are shown in Table 1. This shows that *Invenio* principles are applicable to the 5GC. When 4G EPC and the 5GC coexist, the complexity of manual NF placement further increases.

Proprietary microservices: Microservice architectures use fine-grained autonomous components, fragmenting traditional control-plane NFs into multiple VNFCs [29, 61, 62]. The VNFCs are independently instantiated, and communicate with each other using proprietary message formats. This lack of standardization implies that the roles and functionalities of VNFCs are not well-understood and can change with new versions, altering their affinity. Consequently, service providers must (re)analyze affinity whenever NFs are upgraded or a service is added/removed. Microservices also result in longer, more complex SFCs, reducing the latency allowed for each VNFC [21, 62].

While the lack of standardization can complicate mapping a given message to a user action, microservices often reuse the subscriber-ID/session-ID in traditional signaling protocols [36, 44] to facilitate logging and reduce performance overhead. For example, the timer service (Chronos) in Clearwater [44], a popular microservice-based IMS implementation, uses the "Call-ID" header in Session Initiation Protocol (SIP) messages to manage timers. This behavior can be exploited to trace VNFC-generated messages to user actions.

Lessons learned: The above discussion highlights three consequences for *Invenio*. First, *Invenio* should automate message and event processing, which should be transformed into a protocol-agnostic format before further processing. Second, *Invenio* should understand the relation between different identifiers used by NFs to correlate messages related to the same user. This involves understanding the user-identifying headers used by standard protocols, and correlation of identifiers in proprietary message payloads. Third, *Invenio* should understand user actions and their corresponding responses, and map each message to a specific user action. Since internal implementations of microservice-based systems change frequently, *Invenio* should only use endpoint messages which follow well-known protocols (such as messages (1) and (2) in Fig. 1) to map messages to user actions.

4 PROBLEM DEFINITION

Consider sets N, U, M, and R, where N represents NFs (PNFs, VNFs, or VNFCs) in a network, U represents user devices or end points that utilize the services provided by the network, M represents messages that can be sent or received by all NFs in N, and $R \subseteq M$ represents user request messages generated by a user $u \in U$ and responses sent back to users $\in U$.

To utilize network services, a user $u \in U$ sends a request r to an NF in N. A request sent by an endpoint u triggers the generation of several messages $m \in M$ between a subset of NFs. We denote this subset by N_r. NFs use a number of protocols to complete processing user requests. Typically, an NF will handle a part of the functionality, and forward messages to the next NF in an SFC. Additionally, an NF may utilize interfaces exposed by other NFs to acquire information needed for processing the message itself. Since we aim to model messages that are sent/received by each NF, the manner in which the messages are exchanged is irrelevant and we model messages of all protocols using the set M.

Set $M_r \subset M$ represents the messages triggered to handle a given user request $r \in R$. M_r does not include user-sent or received messages $\in R$. In addition to the messages M_r that are generated by an NF to handle a user-triggered request r, NFs in an SFC may generate messages that are *not* handling a request from u. Such messages include (but are not limited to) messages generated to synchronize state between NFs, and keep-alive or setup/teardown messages. These messages $\in M$ but $\notin M_r$ for any $r \in R$. For example, an online charging message that is generated by an NF to ensure the successful processing of a user-triggered message r is in M_r. An offline charging message that is generated by an NF to later bill the user for an already processed request is $\in M$ but $\notin M_r$.

We use the function Ω to define a mapping between the user-triggered messages and the NFs involved in processing these messages as well as the messages triggered by NFs to handle these messages; that is, $\Omega(r) = (N_r, M_r)$.

A service request from an endpoint may simply include a request message r_{start} from the endpoint and a response r_{end} from the network where $r_{start}, r_{end} \in R$. However, there are cases such as challenge-response procedures where the endpoint may have to respond to multiple requests from network to complete the initial request. That is, the sequence of messages processed by the endpoint to complete a service request is $< r_{start} \cdots r_{end} >$. We use the term "procedure" to refer to this set of messages, R_p, that are exchanged for delivering a specific service to the endpoints, where $R_p = \{r_{start}, \cdots, r_{end}\}$.

For a given procedure p, we define N_p and M_p as follows.

$N_p = \bigcup_{r \in R_p} N_r$ represents all NFs involved in processing messages of procedure R_p.

$M_p = \bigcup_{r \in R_p} M_r$ represents all messages processed by all NFs in N_p to handle messages of procedure R_p.

The NFs and messages involved in processing a procedure are typically the same for every instance of a certain procedure type pt.

Let the $c(pt, n_x, n_y)$ be the number of messages in M_{pt} exchanged between a pair of NFs $n_x, n_y \in N_{pt}$ for procedure type pt. The *procedural affinity* (referred to simply as "affinity" in the remainder of the paper) between NFs n_x, n_y is defined as:

$$\texttt{Affinity}(pt, n_x, n_y) = c(pt, n_x, n_y). \tag{1}$$

Invenio affinity can easily incorporate additional metrics by updating Equation (**1**). For example, the function $c(pt, n_x, n_y)$ can incorporate the number of the hops traversed or latency incurred in communicating with a specific NF or it can avoid over-subscribed links. The function may also be updated to incorporate licensing or hardware constraints that limit the placement of network servers such as the Home Subscriber Server (HSS) and the Online Charging System (OCS).

In summary, *Invenio* computes N_p and M_p, determines the mapping $\Omega(p) = (N_p, M_p)$ for each request r, and identifies each procedure type pt and its associated N_{pt} and M_{pt} sets. This information is then used in Equation (**1**) to compute the affinity between NFs for every procedure type pt, in order to place NFs with relatively high affinity in close proximity. Unlike the work by Sampaio et al. [51], *Invenio* affinity considers the *complete procedure* instead of one or a few messages that are not accurate measures of the entire user experience.

5 *INVENIO* DESIGN

Fig. 3 shows the *Invenio* architecture. The affinity engine is executed after upgrades. A placement engine can use *Invenio* affinity and execute when a new NF is to be instantiated or after major changes in policies or procedure type distribution.

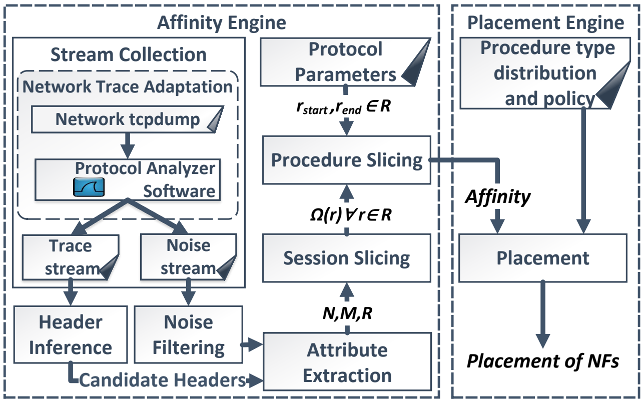


Figure 3: *Invenio* architecture

5.1 Inputs

NF message stream. Invenio uses messages exchanged between NFs in an SFC to identify procedures and their associated messages. Message sequences can be extracted from network traces or NF-provided information such as debug logs or a VNF Event Stream (VES) as specified in ONAP [40] and OPNFV [42]. In the absence of such structured data streams, *Invenio* uses traffic traces from running PNFs/VNFs/VNFCs. These traces can be collected at individual PNFs where physical infrastructure is used, or at Open vSwitch (OVS) or Docker bridge in case VNFs are deployed using virtualization platforms such as OpenStack [41] or Docker [16].

Invenio uses traffic snapshots collected after microservice upgrades. Traffic traces can also be collected during integration tests [28]. The affinity engine merges traffic traces in order to compute affinity values for each procedure type (Equation (**1**)). In the rest of this paper, we use the term **trace stream** to refer to these input traffic snaphots.

Trace streams often contain extraneous messages that are not generated due to endpoint actions and are not part of any M_r. Such messages include setup messages, heartbeat messages, and synchronization messages. These messages are exchanged between NFs even in the absence of user-generated traffic. To eliminate such messages, *Invenio* uses a trace stream collected during an idle period, henceforth referred to as **noise stream**. An idle period is when NFs are running but no traffic is initiated by user devices. For instance, for the Diameter protocol, application-initiated messages such as Session-Termination-Request/Answer (STR/STA) are part of the trace stream, and Device-Watchdog-Request/Answer (DWR/DWA) messages, which used by Diameter protocol to maintain application-level heartbeat between communicating peers, are part of the noise stream.

Invenio utilizes output generated by open-source packet analyzer software such as Wireshark [59] to decode raw messages. We use Wireshark to export Packet Description Markup Language (PDML) and Portal Structure Markup Language (PSML) files, and use these files as inputs. If traffic is encrypted, Wireshark can decrypt it using the keys which will be available to the service provider.

Protocol parameters. Invenio uses developer knowledge specified in a configuration file containing the following information in xml format: **(a) Procedure start/end messages:** are the message pairs $r_{start}, r_{end} \in R$ that are used by endpoints to start and terminate a service request. These values are only required for protocols that are used by endpoints. That is, for the example shown in Fig. 1, these values are only required for the SIP protocol used in the endpoint messages (1) and (2).

(b) Subscriber-ID and session-ID header names: are the names of the headers used by protocols to transmit subscriber-IDs and session-IDs. The headers may be used in conjunction with the output of the header inference module (§5.2) to extract the subscriber-ID and session-ID values from a message. These inputs are only necessary in cases such as the GPRS Tunneling Protocol (GTP), which uses integer identifiers instead of (the more common) text identifiers in the Uniform Resource Identifier format specified in the RFC [9].

Procedure type distribution and policy. Procedure type distribution and provider policy information can decide NF placement. The procedure type distribution can be obtained from an NF that processes endpoint messages. For example, the Proxy Call Session Control Function (P-CSCF) handles all inbound SIP traffic and therefore the P-CSCF NF instance(s) has the procedure type distribution information. Policy information allows providers to optimize NF placement based on their specific requirements (§5.7).

5.2 Header Inference

The first step in generating the set M_{pt} for computing affinity is to find user-identifying headers. A header inference module analyzes messages in the trace stream to identify possible headers that carry the subscriber/session-ID values.

Algorithm 1: Candidate header analysis

```
Input  : $(traceFile, endpointProtocol)$
Output : $candSubHdrN, candSnHdrN$
/* Find candidate user header names for protocol */
1  for each packet p in endpointProtocol do
2      for each header h in p do
3          if h.value conforms to subscriber_id format then
4              $candSubHdrN[p.protocol] \longleftarrow$
                 $candSubHdrN[p.protocol] \cup h.name$
5              $p.candSubV \longleftarrow p.candSubV \cup h.value$
6              $subHdrV = subHdrV \cup (h.value, p.candSubV)$
/* Find candidate user header names for all
   protocols */
7  for each value subV in subHdrV do
8      for each packet p in traceFile do
9          if $p.protocol \neq endpointProtocol$ then
10             for each header h in p do
11                 if $h.value \approx subV$ then
12                     $candSubHdrN[p.protocol] \longleftarrow$
                         $candSubHdrN[p.protocol] \cup h.name$
/* Find candidate session header names */
/* Step 1. Find header values that repeat between
   a pair of NFs */
13 for each protocol protocol in all protocols do
14     for each packet p in protocol do
15         for each header h in p do
16             $uniqueSnV[h.value] \longleftarrow$
                 $uniqueSnV[h.value] \cup (h, p)$
/* Step 2. Eliminate headers whose value repeats
   in messages of multiple users */
17 for each entry e in uniqueSnV do
18     if $|e.p| > 1$ then
19         if $|e.p.candSubV| == 1$ then
20             $candSnHdrN[p.protocol] \longleftarrow$
                 $candSnHdrN[p.protocol] \cup e.h.name$
```

Algorithm 1 computes candidate header names. The algorithm has two stages. Stage 1 (lines 1-12 of Algorithm 1) computes all header names that can carry the subscriber-ID values using the subscriber-ID formats described in [9]. This stage generates a list of candidate subscriber-ID header names (*candSubHdrN*) per protocol. Stage 2 (lines 13-20 of Algorithm 1) identifies all headers whose values repeat in messages exchanged between a pair of NFs using a specific protocol (Step 1, lines 13-16). Since the session-ID headers are used instead of subscriber-ID headers, message exchanges must carry the same value of session-ID in all messages. However, messages exchanged between NFs also carry routing or NF-identifying headers which repeat frequently. These header names are eliminated from the *candSubHdrN* (Step 2, lines 17-20). Table 2 lists sample results that confirm that correct header names were inferred by the algorithm from traces for SIP and Diameter.

Table 2: *Invenio*-generated candidate header names

Protocol	Subscriber-ID		Session-ID		
	Predicted	Actual	Predicted		Actual
			Step-1	Step-2	
SIP	FROM.user, TO.user, contact.user	FROM.user TO.user	R.URI, VIA, Call-ID, CSEQ, ALLOW	Call-ID	Call-ID
Diameter	Service-Ctx-Id User-Name Sub-Id-Data, e1164.msisdn	User-Name, Sub-Id-Data	Origin-Host, Origin-Realm Ssn-ID Auth-App-Id	Ssn-ID	Ssn-ID

5.3 Noise Filtering

The noise filtering module eliminates messages not generated due to user actions that should not be part of M_{pt} for any procedure type. This includes (a) messages exchanged by protocols that are not specified in any procedure start message in the configuration file, (b) messages with type/name matching a message in the noise file, and (c) messages that do not carry any subscriber-ID or session-ID header.

5.4 Attribute Extraction

Since protocols and interfaces use different encoding formats (binary or text) to exchange information, we convert the input message stream to a protocol-agnostic intermediate format: event objects. Each event object is associated with (a) Transport-layer information (source and destination IP address and ports) to identify the NFs in the SFC, and (b) Subscriber-ID and session-ID headers from each packet extracted using the output of header inference module or configuration input.

5.5 Session Slicing

The session slicing module operates on the event objects generated by the attribute extraction module, and uses event information to identify all messages associated with a single user. That is, it computes $\Omega(r) = (N_r, M_r)$ for each user request. This involves correlation of session and subscriber-ID headers from event objects, and identifying all session-IDs that correspond to a single subscriber-ID. Since the input stream may contain messages and events from multiple users, this module analyzes message sequences to find the longest sequence of successful message exchanges, simultaneously merging the shorter sequences due to multiple session/subscriber-IDs.

For example, using the message exchange shown in Fig. 4, the session slicing module will yield an association between the session-IDs of REGISTER and INVITE, as both messages carry the same subscriber-ID. Similarly, while the Re-Authorization-Request does not include a subscriber-ID, it can be associated with the REGISTER/INVITE message in Fig. 4 using the session-ID value from the AA-Request messages, since the INVITE message and the AA-Request carry the same subscriber-ID value, albeit in different headers.

5.6 Procedure Slicing

The procedure slicing module determines the NF set N_{pt} and message set M_{pt} associated with each procedure type tt. The session slicing module gives possible message sequences in $\Omega(r)$. This set can include messages from multiple users, as a single user may not generate all possible procedure types. Since messages in the trace stream are chronologically ordered, the packets in the set M_t corresponding to a certain procedure t have monotonically increasing identifiers. For example, in Fig. 4b, all messages from (7) INVITE to (12) 200 OK are part of the same procedure. Using the r_{start}, r_{end} messages input by service providers in the configuration file, *Invenio* slices messages of each procedure.

Each procedure type pt independently provides a service to a user u. In practice, multiple procedure types may have a strict dependence, and a service may involve invoking multiple procedure types. For example, a voice-call service requires INVITE and BYE as shown in Fig. 4b (messages 7-18). Such procedures are merged in *Invenio*. Sets N_{pt} and M_{pt} are then used to compute the affinity between NFs for every procedure type pt using Equation (**1**).

5.7 Placement

The placement engine can use affinity information generated by the *Invenio* affinity engine, together with input procedure type distribution and provider policies, to make placement decisions. NFs with the highest affinity values (Equation (**1**)) are co-located or placed in close proximity. In our experiments, we identify NFs that must be co-located to minimize the impact of inter-NF latency on interactive services. We will be implementing multi-criteria placement algorithms [29, 51, 61] in future work.

6 EVALUATION

We evaluate *Invenio* with two systems: (a) Clearwater: an open-source microservice-based implementation of IMS, (b) VoLTE: a prototype Voice over LTE (VoLTE) implementation in which NFs are functionally decomposed into microservice-based VNFCs (Fig. 6). We collect network traces from all NFs and use *Invenio* to compute affinity between NFs. We use these affinity values to decide the NF placement and evaluate the performance of two workload types. Our goal is to answer the following questions: (1) How effective is *Invenio* in computing affinity with multiple protocols? (§6.1, §6.2.1) (2) What influence do *Invenio*-generated affinity values have on NF placement? (§6.1, §6.2.2) and (3) What is the impact of inter-NF latency on performance under different workloads? (§6.1, §6.2.3)

Implementation: *Invenio* includes ~2600 lines of Python code. We developed a prototype microservice-based VoLTE system using Kamailio [31] version 5.0.4 as the SIP server for evaluation. We added a REST message interface to Kamailio

Table 3: Testbed configuration

Server	CPU	Cores	RAM	NFs Deployed
R430	2x Intel Xeon E5-2620 v4	16	64 GB	Clearwater
DL120	1x Intel Xeon X3430	4	8 GB	Swarm Workers, Load-Generator

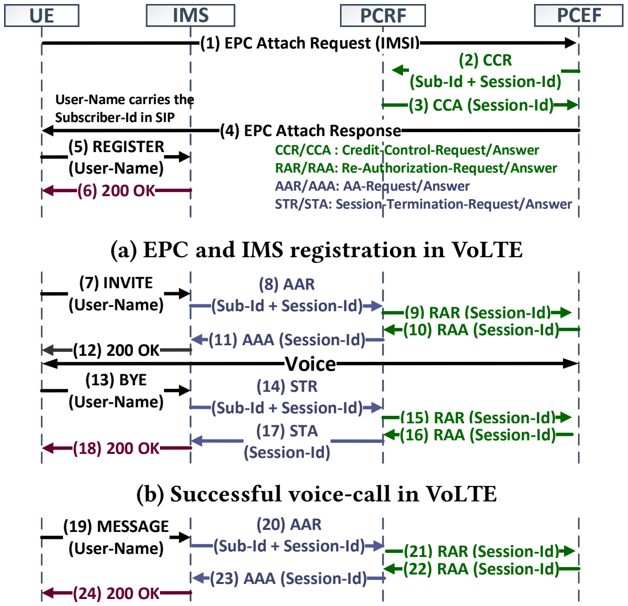
to communicate with the PCRF. We also developed prototype implementations of the PCRF and PCEF for the VoLTE system. All REST-based components are developed as application extensions to the KORE library [57] (version 2.0.0). The PCRF and PCEF are developed as application extensions in the FreeDiameter library [20] version 1.2.1 (~3700 lines of new C code).

Experimental testbed: Our testbed includes one Dell PowerEdge R430 and 5 HP ProLiant DL120 G6 (Table 3) connected by a Gigabit Dell N2024 Switch. We use Docker [16] version 17.03.0-ce and Docker-compose (v1.11.2) to deploy NFs for Clearwater (Fig. 5) and VoLTE (Fig. 6).

Workloads: We use two primary network services: (a) **Voice-call**: This service involves two procedure types (INVITE and BYE). (b) **Short Message Service (text-msg)**: This service utilizes a single procedure of type MESSAGE. We also use SUBSCRIBE (which supports the Presence service) to illustrate the impact of message-count based placement on system performance. However, SUBSCRIBE messages are not generated during performance evaluation and system performance is only evaluated for interactive workloads (voice-call and text-msg).

Following SIP standards, every SIP endpoint registers itself with the IMS network using a REGISTER message (Fig. 4a) before utilizing the voice-call or text-msg service, as depicted in Fig. 4b and 4c. In VoLTE, where the SIP messages are tunneled over the EPC network, a SIP endpoint must additionally attach itself to the EPC network before generating the REGISTER message (steps 1–4 in Fig. 4a). For brevity, we only depict the communication between the IMS and EPC, and omit messages exchanged during the EPC attach [2].

Methodology: SIPp [56] is used to generate two types of workloads: voice-call and text-msg, sending four types of messages: REGISTER, INVITE, BYE and MESSAGE. Each SIPp instance runs on a dedicated physical machine and saturates available system resources. We measure failures by the observing the result code in the SIP response messages. We record the total number of successful calls or messages for each workload type. For the voice-call workload, where multiple procedure types are required to complete a call, we only count the number of calls that were successfully completed; *i.e.*, partially completed calls are ignored. Each experiment runs for 30 seconds. Each experiment is repeated at least 5 times and results are shown with 95% confidence intervals.



(a) EPC and IMS registration in VoLTE

(b) Successful voice-call in VoLTE

(c) Successful text-msg in VoLTE

Figure 4: VoLTE workloads

Affinity computation: The computation is performed offline after system/policy upgrades and therefore does not impact orchestration performance. Since *Invenio* parses each header in each packet for every protocol, the time complexity for affinity computation is O(protocol x packet x header). In practice, however, network traces can be filtered using the IP addresses of NFs, which significantly reduces computation time. Affinity computation takes less than 300 seconds in our experiments with both Clearwater and VoLTE.

6.1 Clearwater Case Study

6.1.1 Architecture. Clearwater [44] is an open-source platform for a microservice-based containerized implementation of an IMS. Clearwater uses REST-based communication to retrieve authentication vectors, manage timers and handle state synchronization, which makes it ideal for a case study. The architecture is illustrated in Fig. 5 (adapted from [44]). Only the components used in our experiments are depicted. We use Clearwater version 1.0 (clearwater-docker release-120). **Bono** an edge proxy that implements the P-CSCF (Proxy Call Session Control Function (CSCF)) in the 3GPP IMS architecture [1]. SIP clients communicate with Bono over UDP/TCP connections. **Sprout** implements the Registrar, I/S-CSCF (Interrogating/Serving CSCF) and Application Server components. **Homestead** provides a REST interface to Sprout for retrieving authentication vectors and user profiles. **Chronos** is a distributed, redundant, reliable timer

Table 4: NF affinity for Clearwater

Procedure Type (pt)	NF Pair (n_x, n_y)	Affinity c(pt, n_x, n_y)
NFs: Bono, Sprout, Ralf		
Voice-call	Bono, Sprout	10
	Bono, Ralf	8
	Sprout, Ralf	4
Text-msg	Bono, Sprout	4
	Bono, Ralf	2
	Sprout, Ralf	0

service. Bono and Sprout report chargeable events to the Charging Trigger Function **Ralf**.

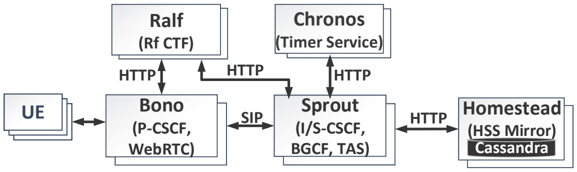


Figure 5: Clearwater architecture

6.1.2 Affinity Analysis. We use *Invenio* to compute affinity between NFs in Clearwater for both voice-call and text-msg workloads. The results are presented in Table 4. We observe that the affinity between Clearwater NFs is different for voice-call and text-msg traffic. For instance, for voice-call traffic, there is high affinity between Bono, Sprout and Ralf, whereas for text-msg traffic, Bono and Ralf only exchange two messages, and no messages are exchanged between Sprout and Ralf. Ralf therefore has a higher affinity with Bono and Sprout for voice-call workload compared to text-msg workload. The placement of Ralf w.r.t. to Bono and Sprout thus has a higher impact on the performance of voice calls compared to the text-msg workload.

6.1.3 Performance. We first benchmark the performance of the voice-call and text-msg workloads with negligible delay. These results serve as baselines and are labeled "ideal" in our plots. We then use "tc" to introduce latency on links connecting two NF pairs (a) Ralf to Sprout and (b) Ralf to Bono, to validate the impact of placement of Ralf on performance. While tail latency values in public clouds typically do no exceed a few ms [14], user mobility in cellular networks can result in deployments where user traffic traverses data centers (or edge and core clouds). Inter-site latency can go up to tens of ms [60], so we experiment with delays up to 20 ms, and compare to the "ideal" case. We make the following observations from our results: (a) Even a single high-latency link can result in significant performance degradation for both voice-call and text-msg workloads, and (b) Performance degradation for the voice-call workload (in which Ralf has higher affinity) is more than for the text-msg workload (in which Ralf has lower affinity). These observations underscore the need for careful VNFC placement. While manual analysis shows that Sprout and Bono (which collectively implement the functionality of the CSCF) must always be co-located, analysis of IMS standards does not suffice for proprietary IMS implementations such as Clearwater in which internal implementation determines the affinity values between VNFCs (Bono/Sprout and Ralf).

6.2 Microservice-based VoLTE

Fig. 6 shows the architecture of a VoLTE system which includes (a) a SIP server that handles SIP/IMS signaling from the endpoints, referred to as the Application Function (AF), (b) a PCRF that allocates QoS rules to a user, (c) a PCEF that enforces QoS rules per user, and (d) a SUB module that provides Presence functionality. The messages exchanged for voice calls and text messages are presented in Fig. 4b and Fig. 4c, respectively.

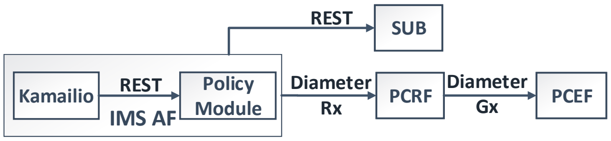


Figure 6: VoLTE system architecture

Our VoLTE implementation can be deployed in multiple configurations and is used to study the impact of microservice decomposition and placement on system performance. In the experiments in the rest of the paper, we treat the VoLTE implementation as a whitebox. That is, we analyze the call flows manually to validate the *Invenio* output. In contrast, the Clearwater IMS implementation was treated as a blackbox where we simply used *Invenio* affinity to study the impact of latency on network services.

Microservice decomposition: We decompose the PCRF into independent microservice components and deploy each microservice as an independent VNFC. Fig. 7 shows the architectures we use to deploy the EPC PCRF. In the legacy PCRF model (monolithic design) depicted in Fig. 7(a) (top left), all components are deployed in a single process and communicate via API calls. This communication is not externally observable, and the entire PCRF is deployed as a single NF. In contrast, in the microservice designs depicted in Fig. 7(b,c,d), the PCRF functionality is collectively provided by the VNFCs described in Table 5. All VNFCs expose a synchronous REST interface for external communication, so communication between the VNFCs is externally observable.

Table 6 compares PCRF µService Design-1, 2 and 3. We observe that in µService Design-1 (Fig. 7(b)), PCRF-Base communicates with all other VNFCs and, consequently, has affinity with all other VNFCs. In µService Design-2 (Fig. 7(c)),

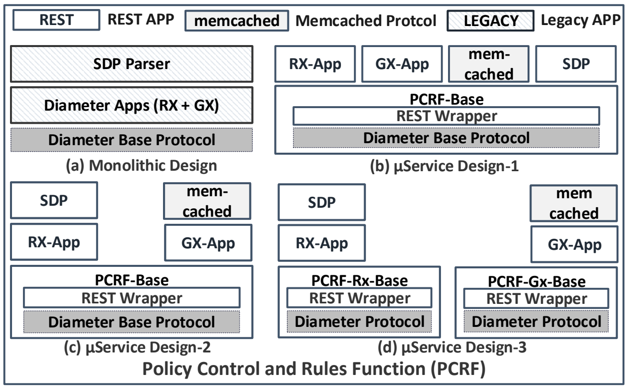


Figure 7: PCRF architectures

Table 5: Functionality of NFs in PCRF

VNF	VNFC	Interface	Functionality
PCRF	PCRF-Base	REST, Rx [5], Gx	Diameter [19] protocol functionality and interface with other VNFCs
	Gx-App	REST	Process Gx Interface messages
	Rx-App	REST	Process Rx Interface messages
	SDP	REST	Process Session Description Protocol (SDP) [27] payload
	PCRF-Gx-Base	Gx, REST	Diameter functionality and interface with other Gx VNFCs
	PCRF-Rx-Base	Rx, REST	Diameter functionality and interface with other Rx VNFCs
	memcached	REST	Gx-App and Rx-App synchronization

the PCRF-Base only has affinity with Rx-App and Gx-App. The SDP VNFC only has affinity with the Rx-App, and the Gx-App has affinity with memcached. In µService Design-3 (Fig. 7(d)), the PCRF-Base VNFC is decomposed into PCRF-Rx-Base and PCRF-Gx-Base, which only have affinity with Rx-App and Gx-App VNFCs, respectively.

Table 6: Manual analysis of communication between VNFCs in PCRF µservice designs

PCRF Decomposition	PCRF Microservices (VNFCs)				
		Gx-App	Rx-App	SDP	memcached
µS Design-1	PCRF-Base	✓	✓	✓	✓
µS Design-2	PCRF-Base	✓	✓	✗	✗
	Gx-App	NA	✗	✓	✓
	Rx-App	✗	NA	✓	✗
µS Design-3	PCRF-Gx-Base	✓	✓	✗	✗
	PCRF-Rx-Base	✗	✓	✗	✗
	Gx-App	NA	✗	✓	✓
	Rx-App	✗	NA	✓	✗

6.2.1 Affinity Analysis. We collect traffic traces (tcpdump) of voice-call and text-msg traffic with PCRF deployed in three configurations (µService Design-1, µService Design-2, and µService Design-3). Table 7 gives the results.

We make two observations from the results: (1) Affinity differs for the voice-call and text-msg traffic. For instance, in µService Design-1, PCRF-Base exchanges two messages with the SDP VNFC in case of voice-call traffic, but the SDP VNFC is not involved in the processing of text-msg traffic, and (2) Affinity differs in the three designs. For instance, for voice-call traffic, in µService Design-2 there is affinity between the PCRF-Base and SDP VNFC. In contrast, in µService Design-2, PCRF-Base only communicates with Rx-App and there is no affinity between the PCRF-Base and SDP VNFC. Affinity is only listed for the monolithic and µService Design-3 for the text-msg workload since others are similar. The SUB VNFC does not involve PCRF, and is omitted from Table 7. We compare *Invenio* results with the results of our manual analysis in Table 5 and verify that *Invenio* accurately identifies the procedures and affinity values for all procedure types.

Table 7: NF affinity for VoLTE

Procedure Type (pt)	NF Pair (n_x, n_y)	Affinity c(pt, n_x, n_y)
Monolithic Design, NFs: AF,PCRF,PCEF		
Voice-call	AF, PCRF	4
	PCRF, PCEF	4
Text-msg	AF, PCRF	2
	PCRF, PCEF	2
µService Design-1, AF, PCRF-Base, Gx-App, Rx-App, SDP, memcached, PCEF		
Voice-call	AF, PCRF-Base	4
	PCRF-Base, Gx-App	4
	PCRF-Base, Rx-App	4
	PCRF-Base, SDP	2
	PCRF-Base, memcached	4
	PCRF-Base, PCEF	4
µService Design-2, AF, PCRF-Base, Gx-App, Rx-App, SDP, memcached, PCEF		
Voice-call	AF, PCRF-Base	4
	PCRF-Base, Gx-App	4
	PCRF-Base, Rx-App	4
	Gx-App, memcached	4
	Rx-App, SDP	2
	PCRF-Base, PCEF	4
µService Design-3, AF, PCRF-Rx-BASE, PCRF-Gx-Base, Gx-App, Rx-App, SDP, memcached, PCEF		
Voice-call	AF, PCRF-Rx-Base	4
	PCRF-Rx-Base, Rx-App	2
	PCRF-Rx-Base, PCRF-Gx-Base	2
	Rx-App, PCRF-Gx-Base	2
	Rx-App, SDP	2
	PCRF-Gx-Base, Gx-App	4
	Gx-App, memcached	4
	PCRF-Gx-Base, PCEF	4
Text-msg	AF, PCRF-Rx-Base	2
	Rx-App, PCRF-Gx-Base	2
	PCRF-Rx-Base, Rx-App	2
	PCRF-Gx-Base, Gx-App	2
	Gx-App, memcached	2
	Rx-App, SDP	0
	PCRF-Gx-Base, PCEF	2

6.2.2 Placement. To study the impact of *Invenio*-generated affinity values on placement, we deploy VoLTE with PCRF µService Design-3 on a Docker Swarm [18] cluster with three nodes. Each node is allocated a maximum of 4 VNFCs by the orchestrator. Fig. 8(a) (top) shows an ideal placement on this cluster. AF is deployed as two VNFCs (Kamailio and Policy-Module), which are always co-located, so we show it

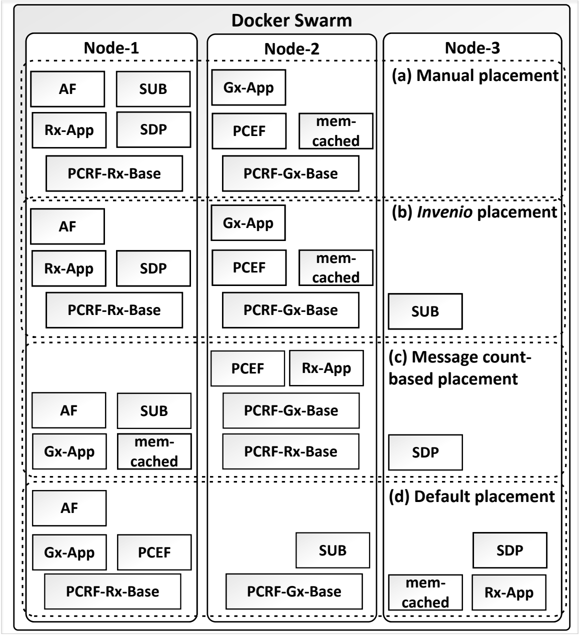


Figure 8: VoLTE placements on three machines

as AF. Fig. 8(b) shows placement with *Invenio*-generated affinity values. The affinity values and resulting constraints are given to the Swarm orchestrator by the "affinity_group" [17] parameter in the Docker-compose configuration file.

Fig. 8(c) shows the result of an instantiation in which the number of messages exchanged between VNFCs is used to decide placement, as discussed by Sampaio et al. [51]. This results in a placement where VNFCs that exchange the highest number of messages (AF, SUB, and memcached, seen on the left side of Fig. 2) are co-located. Any procedure type distribution which has at least 75% presence traffic (lower than the 90% in busy-hour IMS traffic in [7]) will yield the same placement. Fig. 8(d) depicts the results of an instantiation with no constraints given to the Swarm orchestrator, yielding a random NF placement (labeled "Default" in Fig. 8).

We note here that our placement decisions are only based on affinity values and procedure type distribution, and we do not consider extrinsic factors such as network policy, hardware constraints, or link usage in these placement decisions. Such additional metrics can be incorporated into placement decisions by updating Equation (**1**) as described in §4.

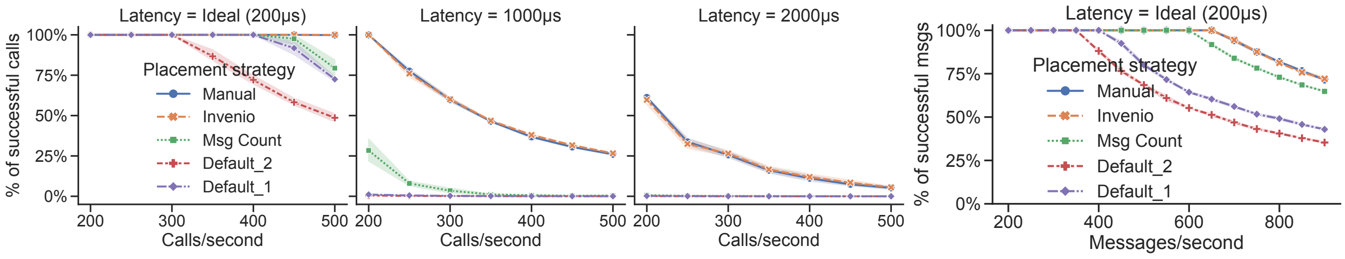
6.2.3 Performance. We evaluate the performance of the four placement strategies shown in Fig. 8 using the two workloads (voice-call and text-msg) and three different inter-NF latencies, where the Docker worker nodes are connected by 200 μs (ideal), as well as 1000 μs and 2000 μs links, which are chosen to emulate inter-rack tail latencies in public clouds [24]. Fig. 9 shows the results of two different outcomes of the default placement strategy – labeled Default 1 (which corresponds to Fig. 8(d)) and Default 2 with (PCEF, PCRF-RX) on Node-1, (AF, SUB, SDP, memcached) on Node-2, and (PCRF-GX, GX-App, Rx-App) on Node-3. We choose PCRF μService Design-3 here as it completely decomposes the NFs into constituent microservices and therefore it is ideal for demonstrating the impact of placement.

Fig. 9a and 9b show that (1) *Invenio* closely matches the results of manual (handcrafted) placement for both voice-calls and text-msg workloads. The results of text-msg workload with 1000 μs, and 2000 μs follow similar trends as the voice-call workloads, and are omitted for brevity, (2) Both message count-based and default (random) placement strategies experience significant performance degradation as the inter-worker (inter-rack) latency increases. The impact of latency is insignificant at lower call rates, but there is significant drop in the overall system throughput as the call rate approaches system capacity, (3) The performance degradation for the voice-call workload is higher than the performance degradation for the text-msg workload. For example, comparing default placement under ideal conditions (inter-rack latency of 200 μs) to *Invenio* placement, for voice-call workload of 500 calls/second, nearly 52% calls are dropped but for text-msg workload of 900 calls/second, less than 36% of calls are dropped. This is a consequence of higher affinity between the NFs shown in Table 7. (4) Default placement – as seen from the results of Default 1 and Default 2 – can lead to significant variation in system performance, complicating capacity planning for microservice-based applications. Placement with *Invenio* clearly outperforms count-based and default placement while avoiding the non-deterministic outcomes inherent to count-based and default placement.

7 OTHER APPLICATIONS

In addition to its use for affinity computation, *Invenio* can be extended for interoperability checking and fault diagnosis.

NF interoperability: The *Invenio* "Session Slicing" module (§5.5) outputs the sequence of messages processed by individual NFs, which can be used to construct a partial state machine. This state machine can be used to check interoperability between NFs from different vendors or of different versions. As an example, we used *Invenio* to check interoperability between the Home Subscriber Server (HSS) in Openair-cn [39] and the Mobility Management Entity (MME) in OpenEPC [12]. We configured *Invenio* to extract Diameter [19] messages exchanged between the MME and HSS in the network traces collected from OpenEPC and Openair-cn. By comparing these messages, we found that the OpenAir HSS expects a Session-Id Attribute Value Pair (AVP) in all messages, which is not provided by the OpenEPC MME. Proprietary message formats can also be supported. We developed wireshark dissectors for the EPC implementation



(a) Voice-call performance (200 μs, 1000 μs, and 2000 μs inter-machine latency)

(b) Text-msg performance

Figure 9: Impact of latency and affinity on VoLTE performance

in [30], which uses proprietary message formats, and successfully used *Invenio* to compare the state machine of this implementation with NFs in Openair-cn.

Fault diagnosis: *Invenio* can be used to diagnose scenarios when configuration or implementation problems at a specific NF result in service interruptions (failures or delays) for one or more users. In production environments where NFs process thousands of messages per second, identifying which NF in an SFC triggered a problem can be tedious and time-consuming, and requires a service provider to understand all protocols and message formats. An important step in identifying failures for a specific user is isolating messages (across an entire SFC) associated with a given user. This can be achieved by the "Session Slicing" module. *Invenio* can then report failed procedures and associated messages M_T.

8 RELATED WORK

Service placement and monitoring: Functionality-based decomposition has been proposed to reduce latency and increase throughput for cellular network control planes [32, 45, 48, 55]. That work uses manual analysis of network architecture and traffic to find the functional elements that can be aggregated. Stratos [22] avoids traversing oversubscribed inter-rack links during function placement, and Selimi et al. [52] explore placement to maximize bandwidth utilization. None of these studies consider workload types and procedures. Other work [29, 51, 61] formulates the placement problem as a graph partitioning problem or an optimization problem. This is orthogonal to our work, as our notion of procedural affinity is a new input to the placement problem.

Recent studies [11, 37, 38, 43, 62] have highlighted challenges in deploying and managing microservice-based applications. For instance, ucheck [43] uses runtime verification and enforcement of invariants to aid service providers. Probius [38] finds performance bottlenecks by correlating VNF, hypervisor, and system metrics. NFVPerf [37] uses network traces to compute per-hop message processing latency which is then used to infer performance bottlenecks. Our work proposes procedure-driven microservice deployment, and is thus complementary to this line of work.

Traffic enrichment: Several efforts [25, 46] enrich network messages with "Metadata" [25] using the Network Service Header (NSH [46]).*Invenio* can use metadata carried in these headers to identify the user associated with a message.

Protocol inference: Extracting protocol state information from network traces, or *reverse engineering* a protocol, has been studied in the literature. Prior work extracts specifications of unknown protocols [8, 10, 13, 33, 58], and uses inferred message formats to detect malware signatures [35, 58]. *Invenio* also exploits protocol header information, but extracts user-identifying headers. Other categories of work in this area use xml or json formats to derive protocol state machines [23, 47]. Application-specific information is used to group messages into sessions [23]. Network traces identify dialogs in HTTP and SIP traffic [47]. While *Invenio* shares finite-state machine extraction techniques with these papers, it differs in one important aspect: we use the extracted state information to compute affinity between NFs for an entire SFC. In contrast, prior work extracts the state machine for a single NF and does not merge state machines from multiple NFs to derive procedure information for an SFC.

9 CONCLUSIONS

In this paper, we have presented *Invenio*, a system that helps service providers to better manage the ever-growing complexity of microservice-based network functions (NFs). We showed that *Invenio* can automatically compute the correct communication affinity between NFs for each user-triggered procedure. This allows service providers to make and update NF placement decisions without the time-consuming and error-prone manual analysis currently used. Our experiments with IMS and VoLTE implementations confirm that, by using procedural affinity-based NF placement, service providers can effectively support services with stringent latency requirements. We expect automatic communication affinity computation for each user action to significantly grow in importance in the coming years as more complex disaggregated services are deployed [3, 21].

REFERENCES

[1] 3GPP. *TS 23.228, IP Multimedia Subsystem (IMS).* http://www.3gpp.org/DynaReport/23228.htm.

[2] 3GPP. *TS 23.401, GPRS Enhancements for Evolved Universal Terrestrial Radio Access Network (E-UTRAN) Access.* http://www.3gpp.org/ftp/Specs/html-info/23401.htm.

[3] 3GPP. *TS 23.501, System Architecture for the 5G System.* http://www.3gpp.org/ftp/Specs/html-info/23501.htm.

[4] 3GPP. *TS 29.212, Policy and charging control over Gx reference point (Release 14).* http://www.3gpp.org/DynaReport/29212.htm.

[5] 3GPP. *TS 29.214, Policy and charging control over Rx reference point (Release 14).* http://www.3gpp.org/DynaReport/29214.htm.

[6] 3GPP. *TS 29.229, Cx and Dx interfaces based on the Diameter protocol (Release 14).* http://www.3gpp.org/DynaReport/29229.htm.

[7] Abhayawardhana, V. S., and Babbage, R. A traffic model for the IP multimedia subsystem (IMS). In *2007 IEEE 65th Vehicular Technology Conference - VTC2007-Spring* (2007), pp. 783–787.

[8] Antunes, J., Neves, N., and Verissimo, P. Reverse engineering of protocols from network traces. In *2011 18th Working Conference on Reverse Engineering* (Oct 2011), pp. 169–178.

[9] Berners-Lee, T., Fielding, R. T., and Masinter, L. Uniform Resource Identifier (URI): Generic Syntax. STD 66, RFC Editor, January 2005. http://www.rfc-editor.org/rfc/rfc3986.txt.

[10] Bossert, G., Guihéry, F., and Hiet, G. Towards automated protocol reverse engineering using semantic information. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security* (New York, NY, USA, 2014), ASIA CCS '14, ACM, pp. 51–62.

[11] Cerny, T., Donahoo, M. J., and Trnka, M. Contextual understanding of microservice architecture: Current and future directions. *SIGAPP Appl. Comput. Rev. 17*, 4 (Jan. 2018), 29–45.

[12] Core Network Dynamics. *OpenEPC - Evolved Packet Core (vEPC).* https://www.openepc.com/.

[13] Cui, W., Kannan, J., and Wang, H. J. Discoverer: Automatic protocol reverse engineering from network traces. In *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium* (Berkeley, CA, USA, 2007), SS'07, USENIX Association, pp. 14:1–14:14.

[14] Dean, J., and Barroso, L. A. The tail at scale. *Commun. ACM 56*, 2 (feb 2013), 74–80.

[15] Dialogic. *IMS and VoLTE 3GPP Interfaces.* https://edit.dialogic.com/edu/interfaces.

[16] Docker. *Build, Ship, and Run Any App, Anywhere.* https://www.docker.com.

[17] Docker. *Docker Compose.* https://docs.docker.com/compose/.

[18] Docker. *Docker Swarm.* https://www.docker.com/products/docker-swarm.

[19] Fajardo, V., Arkko, J., Loughney, J., and Zorn, G. Diameter base protocol. RFC 6733, RFC Editor, October 2012. http://www.rfc-editor.org/rfc/rfc6733.txt.

[20] freeDiameter. *Diameter Open Implementation.* http://www.freediameter.net/trac/.

[21] Gan, Y., Zhang, Y., Cheng, D., Shetty, A., Rathi, P., Katarki, N., Bruno, A., Hu, J., Ritchken, B., Jackson, B., Hu, K., Pancholi, M., Clancy, B., Colen, C., Wen, F., Leung, C., Wang, S., Zaruvinsky, L., Espinosa, M., He, Y., and Delimitrou, C. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud and Edge Systems. In *Proc. of ASPLOS* (April 2019).

[22] Gember, A., Grandl, R., Anand, A., Benson, T., and Akella, A. Stratos: Virtual middleboxes as first-class entities. Tech. rep., University of Wisconsin-Madison, 2012. Technical report TR1771.

[23] Griffeth, N., Cantor, Y., and Djouvas, C. Testing a network by inferring representative state machines from network traces. In *Software Engineering Advances, International Conference on* (Oct 2006), pp. 31–31.

[24] Guo, C., Yuan, L., Xiang, D., Dang, Y., Huang, R., Maltz, D., Liu, Z., Wang, V., Pang, B., Chen, H., Lin, Z.-W., and Kurien, V. Pingmesh: A large-scale system for data center network latency measurement and analysis. *SIGCOMM Comput. Commun. Rev. 45*, 4 (aug 2015), 139–152.

[25] Halpern, J., and Pignataro, C. Service function chaining (sfc) architecture. RFC 7665, RFC Editor, October 2015.

[26] Han, B., Gopalakrishnan, V., Ji, L., and Lee, S. Network function virtualization: Challenges and opportunities for innovations. *IEEE Communications Magazine 53*, 2 (2015), 90–97.

[27] Handley, M., Jacobson, V., and Perkins, C. Sdp: Session description protocol. RFC 4566, RFC Editor, July 2006. http://www.rfc-editor.org/rfc/rfc4566.txt.

[28] Hao, R., Lee, D., Sinha, R. K., and Griffeth, N. Integrated system interoperability testing with applications to VoIP. *IEEE/ACM Transactions on Networking (TON) 12*, 5 (2004), 823–836.

[29] Hu, Y., de Laat, C., and Zhao, Z. Optimizing service placement for microservice architecture in clouds. *Applied Sciences 9*, 21 (2019).

[30] Jain, A., , Lohani, S. K., and Vutukuru, M. A comparison of SDN and NFV for re-designing the LTE packet core. In *2016 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)* (Nov 2016), pp. 74–80.

[31] Kamailio. *Kamailio SIP Server.* https://www.kamailio.org/w/.

[32] Katsalis, K., Nikaein, N., Schiller, E., Favraud, R., and Braun, T. I. 5G architectural design patterns. In *2016 IEEE International Conference on Communications Workshops (ICC)* (May 2016), pp. 32–37.

[33] Krueger, T., Krämer, N., and Rieck, K. ASAP: Automatic semantics-aware analysis of network payloads. In *Proceedings of the International ECML/PKDD Conference on Privacy and Security Issues in Data Mining and Machine Learning* (Berlin, Heidelberg, 2011), PSDML'10, Springer-Verlag, pp. 50–63.

[34] Kubernetes. *Kubernetes.* https://kubernetes.io/.

[35] Leita, C., Mermoud, K., and Dacier, M. Scriptgen: an automated script generation tool for honeyd. In *21st Annual Computer Security Applications Conference (ACSAC'05)* (Dec 2005), pp. 12 pp.–214.

[36] Microsoft. *Designing microservices: Logging and monitoring*, 2018. https://docs.microsoft.com/en-us/azure/architecture/microservices/logging-monitoring.

[37] Naik, P., Shaw, D. K., and Vutukuru, M. NFVPerf: Online performance monitoring and bottleneck detection for NFV. In *2016 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)* (Nov 2016), pp. 154–160.

[38] Nam, J., Seo, J., and Shin, S. Probius: Automated approach for VNF and service chain analysis in software-defined NFV. In *Proceedings of the Symposium on SDN Research* (New York, NY, USA, 2018), SOSR '18, ACM, pp. 14:1–14:13.

[39] OAI / Openair-cn · GitLab. *Evolved Core Network Implementation of OpenAirInterface.* https://gitlab.eurecom.fr/oai/openair-cn.

[40] ONAP. *ONAP - Open Networking Automation Platform.* https://www.onap.org/.

[41] OpenStack. *Open source software for creating private and public clouds.* https://www.openstack.org/.

[42] OPNFV. *OPNFV: Open Platform for NFV*, 2018. https://www.opnfv.org/.

[43] Panda, A., Sagiv, M., and Shenker, S. Verification in the age of microservices. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems* (New York, NY, USA, 2017), HotOS '17, ACM, pp. 30–36.

[44] Project Clearwater. *IMS in cloud.* http://www.projectclearwater.org/.

[45] Qazi, Z. A., Walls, M., Panda, A., Sekar, V., Ratnasamy, S., and Shenker, S. A high performance packet core for next generation cellular networks. In *Proceedings of the Conference of the ACM Special*

Interest Group on Data Communication (New York, NY, USA, 2017), SIGCOMM '17, ACM, pp. 348–361.

[46] Quinn, P., Elzur, U., and Pignataro, C. Network service header (nsh). RFC 8300, RFC Editor, January 2018.

[47] Rafique, M. Z., Caballero, J., Huygens, C., and Joosen, W. Network dialog minimization and network dialog diffing: Two novel primitives for network security applications. In *Proceedings of the 30th Annual Computer Security Applications Conference* (New York, NY, USA, 2014), ACSAC '14, ACM, pp. 166–175.

[48] Raza, M. T., Kim, D., Kim, K. H., Lu, S., and Gerla, M. Rethinking LTE network functions virtualization. In *2017 IEEE 25th International Conference on Network Protocols (ICNP)* (Oct 2017), pp. 1–10.

[49] Raza, M. T., and Lu, S. Enabling low latency and high reliability for IMS-NFV. In *2017 13th International Conference on Network and Service Management (CNSM)* (Nov 2017), pp. 1–9.

[50] Rosenberg, J., Schulzrinne, H., Camarillo, G., Johnston, A., Peterson, J., Sparks, R., Handley, M., and Schooler, E. SIP: Session initiation protocol. RFC 3261, RFC Editor, June 2002. http://www.rfc-editor.org/rfc/rfc3261.txt.

[51] Sampaio, A. R., Rubin, J., Beschastnikh, I., and Rosa, N. S. Improving microservice-based applications with runtime placement adaptation. *Journal of Internet Services and Applications 10* (2019).

[52] Selimi, M., Cerdà-Alabern, L., Sánchez-Artigas, M., Freitag, F., and Veiga, L. Practical service placement approach for microservices architecture. In *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)* (2017), pp. 401–410.

[53] Sharma, S., Miller, R., and Francini, A. A cloud-native approach to 5G network slicing. *IEEE Communications Magazine 55*, 8 (2017), 120–127.

[54] Sheoran, A., Fahmy, S., Osinski, M., Peng, C., Ribeiro, B., and Wang, J. Experience: Towards automated customer issue resolution in cellular networks. In *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking* (2020), MobiCom '20.

[55] Sheoran, A., Sharma, P., Fahmy, S., and Saxena, V. Contain-ed: An NFV micro-service system for containing e2e latency. In *Proceedings of the Workshop on Hot Topics in Container Networking and Networked Systems* (2017), HotConNet '17, pp. 12–17.

[56] SIPp. *Welcome to SIPp.* http://sipp.sourceforge.net/.

[57] Vink, J. *KORE.io - An easy to use web platform for C.* https://kore.io/.

[58] Wang, Y., Zhang, Z., Yao, D. D., Qu, B., and Guo, L. Inferring protocol state machine from network traces: A probabilistic approach. In *Proceedings of the 9th International Conference on Applied Cryptography and Network Security* (Berlin, Heidelberg, 2011), ACNS'11, Springer-Verlag, pp. 1–18.

[59] Wireshark. *Wireshark Go Deep.* https://www.wireshark.org.

[60] Xu, M., Fu, Z., Ma, X., Zhang, L., Li, Y., Qian, F., Wang, S., Li, K., Yang, J., and Liu, X. From cloud to edge: A first look at public edge platforms. In *Proceedings of the 21st ACM Internet Measurement Conference* (New York, NY, USA, 2021), IMC '21, Association for Computing Machinery, p. 37–53.

[61] Yu, Y., Yang, J., Guo, C., Zheng, H., and He, J. Joint optimization of service request routing and instance placement in the microservice system. *Journal of Network and Computer Applications 147* (2019), 102441.

[62] Zhang, Y., Hua, W., Zhou, Z., Suh, G. E., and Delimitrou, C. Sinan: ML-based and QoS-aware resource management for cloud microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (2021), ASPLOS 2021, p. 167–181.