December 2021

CONTAINERIZED DEPLOYMENT OF WEBRTC-SIP INTERWORKING FUNCTION TO INTEROPERATE WITH LEGACY SIP LINE-SIDE EDGES

Faisal Siyavudeen

Ram Mohan R

Sachin Mehrotra

Krishna Kumar Rai

CONTAINERIZED DEPLOYMENT OF WEBRTC-SIP INTERWORKING FUNCTION TO INTEROPERATE WITH LEGACY SIP LINE-SIDE EDGES

AUTHORS:
Faisal Siyavudeen
Ram Mohan R
Sachin Mehrotra
Krishna Kumar Rai

ABSTRACT

Conventional Session Initiation Protocol (SIP) line-side edges are not always distributed, and they require that registrations and calls be handled by the same entity. Servers that support Web Real-Time Communication (WebRTC) clients do not require a hard state and with cloud deployments they are increasingly being deployed as containerized workloads. Containerized deployments (such as Kubernetes) are typically stateless, and even with stateful implementations special handling is required to ensure that registrations and calls are consistently sent to the same SIP edge node, with high availability, in the face of frequent pod failures. To address such challenges, techniques are presented herein that enable a browser (that is stateless) to register via a Kubernetes cluster of pods (which are, again, stateless) but still connect as a SIP line side to a legacy SIP system that requires stickiness in terms of using the same Transmission Control Protocol (TCP) or Transport Layer Security (TLS) connection for SIP registrations and calls.

DETAILED DESCRIPTION

Conventional Session Initiation Protocol (SIP) line-side edges are not always distributed, and they require that registrations and calls are handled by the same entity. Typical examples include a unified border element, collaboration gateway solutions, etc. Servers that support Web Real-Time Communication (WebRTC) clients do not require a hard state and with cloud deployments they are increasingly being deployed as containerized workloads. Containerized deployments (such as Kubernetes) are typically stateless, and even with stateful implementations special handling is required to ensure that

registrations and calls are consistently sent to the same SIP edge node, with high availability, in the face of frequent pod failures.

Techniques are presented herein that ensure that WebRTC-SIP interworking with legacy SIP edges will correctly route traffic while running in container platforms such as Kubernetes. Aspects of the presented techniques (which will be described an illustrated in the narrative that is presented below) support, among other things, a unique way of joining browser devices that can move across pods to still be able to associate with the same user and register or deregister accordingly. For example, when a browser that was initially registered with Pod1 moves to Pod2 such a move will trigger a replacement of the previous registration with the current one. Aspects of the techniques presented herein allow various call features like hold and resume, transfer, call forward, call park and pickup, etc. to work seamlessly across browser endpoints within a SIP ecosystem even when a browser is moving across various pods. Such support is accomplished by, for example, check pointing the context.

Figure 1, below, presents an exemplary overall architecture using Kubernetes.

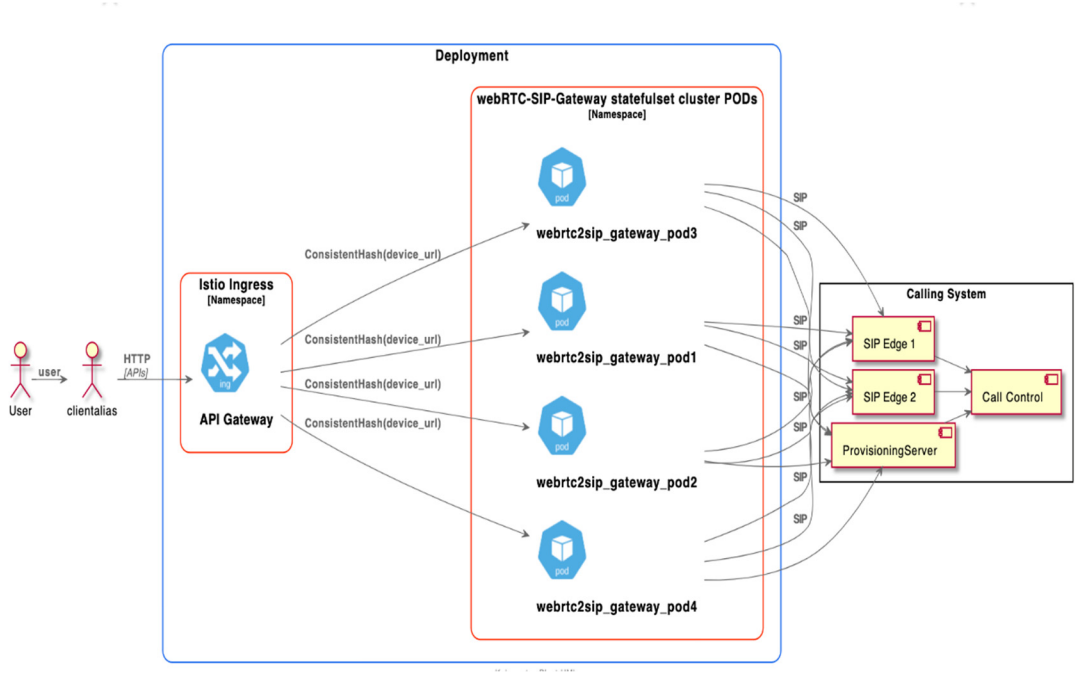

Figure 1: Exemplary Architecture

The WebRTC to SIP Gateway pod cluster that is depicted in Figure 1, above, performs WebRTC Hypertext Transfer Protocol (HTTP) to SIP line side interworking and provides a rich feature set. The WebRTC to SIP Gateway can be a gateway that acts as a line side edge to allow browsers to register as a SIP line to legacy SIP system or can be a gateway that allows browsers to call a SIP system device via a trunk as a guest caller. Some of the supported flows include the registration of a browser as a SIP line side to a SIP cloud calling or legacy SIP enterprise ecosystem, the ability to make inbound or outbound calls, the ability to trigger various supplementary services (such as hold, resume, transfer, call forward, park or resume, etc.), and the ability to perform any of the above-described actions when a browser moves across pods.

Aspects of the techniques presented herein support a number of functionalities. For example, a specific pod will contain the WebRTC-SIP interworking instances. These may be deployed in a single container or they may be separate. During operation of the system as illustrated in Figure 1, a client device will send a Device identifier (ID) in a custom header or body in all of the application programming interface (API) calls that are invoked by the client. The device ID will be subjected to consistent hashing (through, for example, a hash ring or MagLev) to load balance requests across different pods. Since a Device ID does not change, all of the requests from a given device will always be routed to the same pod.

Pods will be created (as, for example, a Kubernetes StatefulSet) such that each pod may be specifically named. Every device that is created on a given pod will result in a SIP REGISTER request from that pod to the destination legacy SIP edge. All of the calls to and from the corresponding client device will use the same Transmission Control Protocol (TCP) connection between that same pod and SIP edge, thereby satisfying the SIP edge requirement for stickiness between registrations and calls and ensuring that all of the messages in a call are carried over the same TCP connection. The SIP REGISTER message will contain a unique identifier (which is a Device ID) that will ensure that any pod that handles the API registration call from a browser will end up registering the same device (essentially replacing the previous registration). Figure 2, below, presents an exemplary flow of the techniques of this proposal.

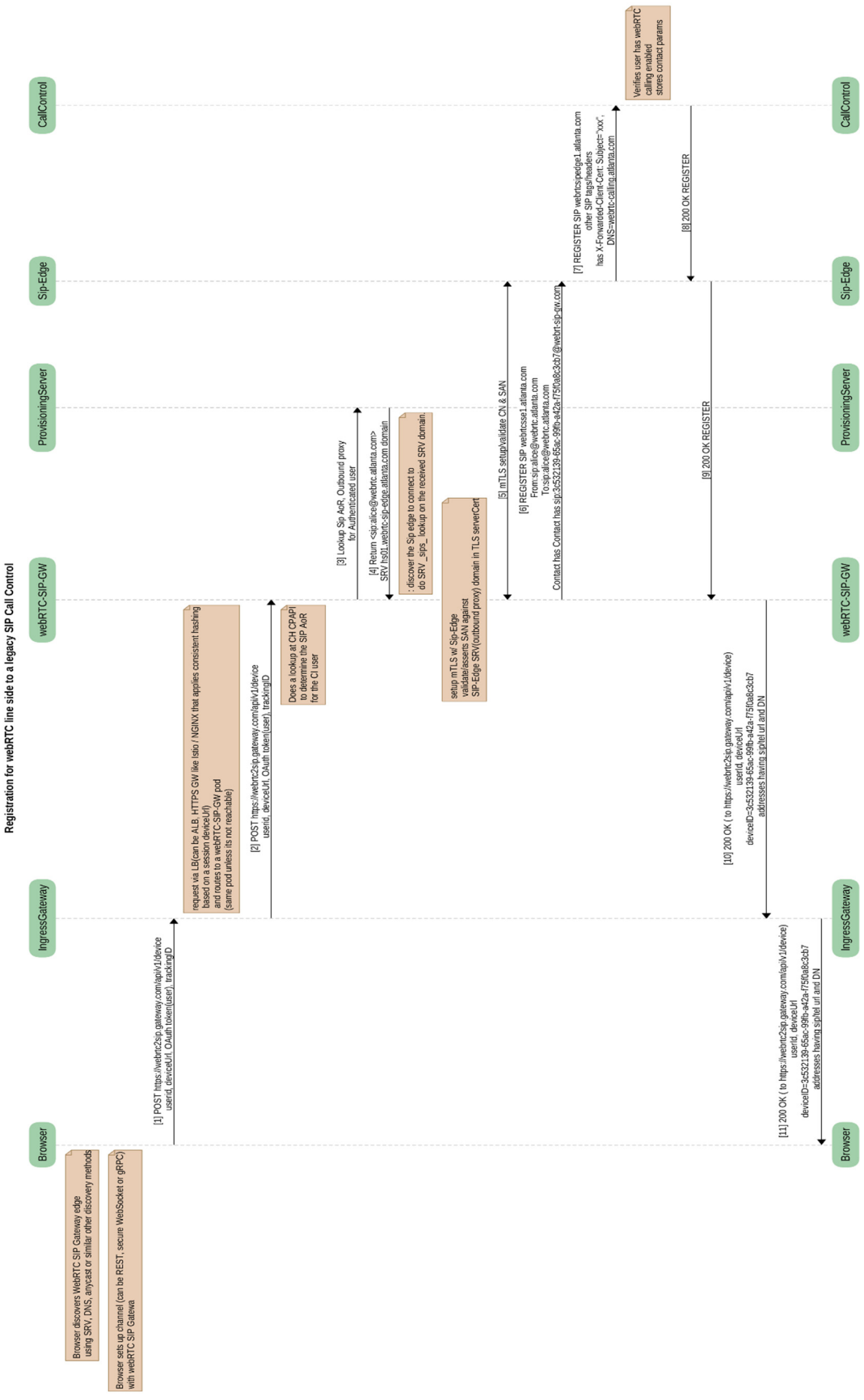

Figure 2: Exemplary Flow

As shown in Step 5 of the flow that is presented in Figure 2, above, a Contact header will contain a unique Device ID that is generated for each browser session and such a value will be used in the reverse direction as well to map the browser.

The failover design as proposed herein ensures that devices can failover at any point in a call. This includes the failover of a WebRTC to SIP Gateway pod during registration, in an active call state, during call transit (e.g., hold, transfer, park, etc.), etc. Figure 3, below, presents an example of how registration and call failover (on call keep-alive) may be performed from one pod to another pod.

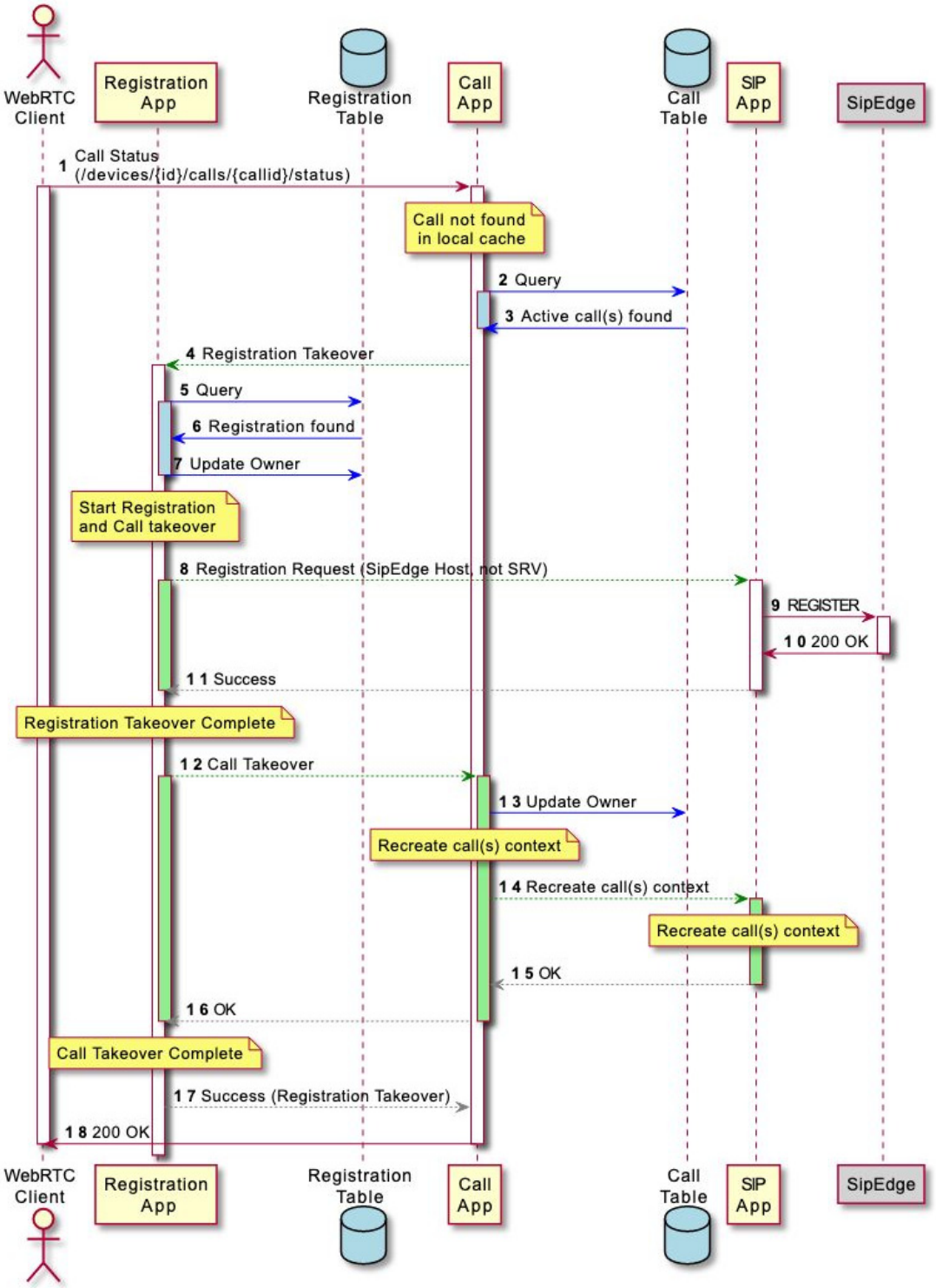

Figure 3: Exemplary Registration and Call Failover

Aspects of the techniques presented herein support a number of capabilities. A first capability encompasses a method whereby a browser inserts a unique Device ID (e.g., that is unique to a login) into a HTTP header or a body which in turn is used in the SIP Contact header of a REGISTER request to uniquely identify a registration session of a user. The unique Device ID (e.g., a random universally unique identifier (UUID)) is mapped to a legacy SIP user (e.g., a SIP Address of Record (AOR)) for that device type.

A second capability encompasses a method whereby the unique Device ID is used to map all of the (stateless) HTTP requests to the same pod (e.g., using StatefulSets) and wherein that pod ensures that it reuses a directed SIP line side connection to a legacy SIP edge. The pod in turn ensures that it takes over the registration (effectively replacing the registration on the SIP side) if the consistent hash routes the API call to a new pod (e.g., due to failover of another pod).

A third capability encompasses a method whereby the pods ensure that call context is stateless on the HTTP side allowing browser clients to move across pods or connect to different pods (when, for example, a pod that setup a call fails over) but ensuring the call is stateful on the SIP leg. This is done by check pointing the minimal call context in a shared database which in turn may be used by any pod to reconstruct the call state and ensure that the SIP peer on other side still sees the same call.

A fourth capability encompasses a method whereby various supplementary calling services (such as hold, resume, call park or retrieve, transfer steps) may be done such that pod failures during an operation still ensure that functionality does not fail. For example, a browser may place a call on hold through POD1 and then get it retrieved through POD2.

In summary, techniques have been presented herein that enable a browser (that is stateless) to register through a Kubernetes cluster of pods (which are, again, stateless) but still connect as a SIP line side to a legacy SIP system that requires stickiness in terms of using the same TCP or Transport Layer Security (TLS) connection for SIP registrations and calls.