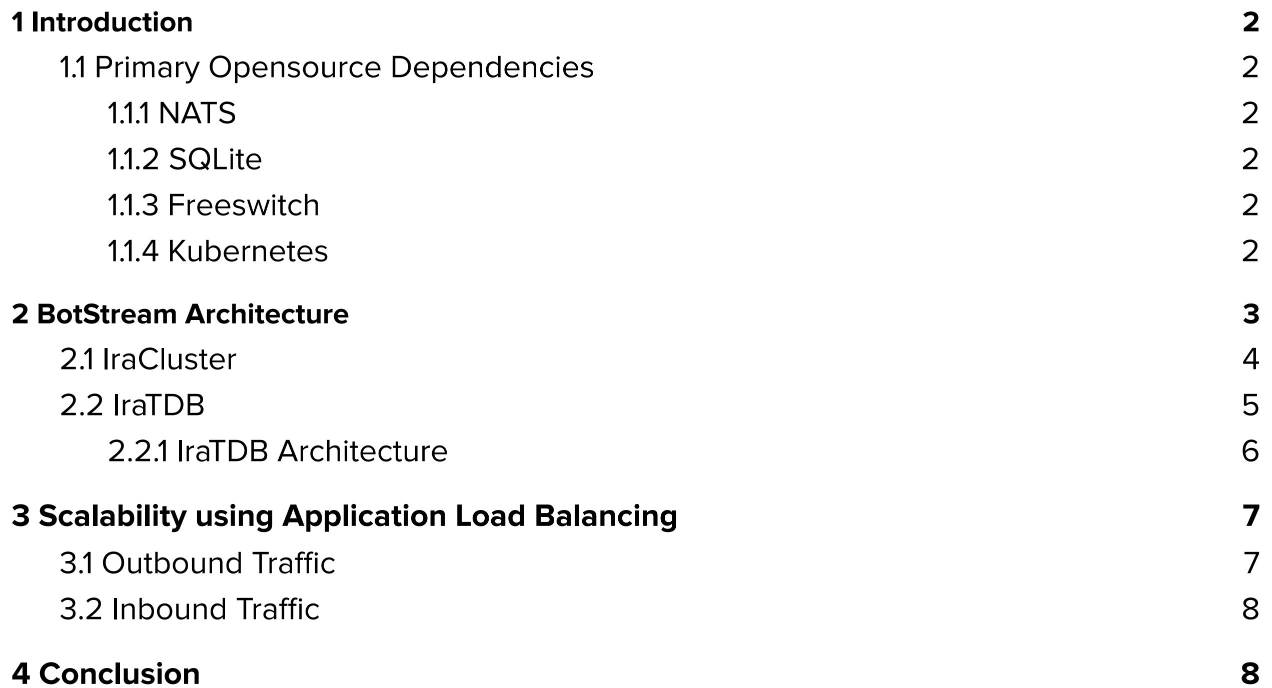
EPICODE

WhitePaper

BotStream

Epicode Design Team

16th May, 2022



1 Introduction **2**
1.1 Primary Opensource Dependencies 2
1.1.1 NATS 2
1.1.2 SQLite 2
1.1.3 Freeswitch 2
1.1.4 Kubernetes 2
2 BotStream Architecture **3**
2.1 IraCluster 4
2.2 IraTDB 5
2.2.1 IraTDB Architecture 6
3 Scalability using Application Load Balancing **7**
3.1 Outbound Traffic 7
3.2 Inbound Traffic 8
4 Conclusion **8**

1 Introduction

BotStream is a bidirectional voice streamer that interfaces telephony services with conversational voice AI applications. It includes an API Dialer, Call recording, Trunk manager, QOS monitor and CPA. It supports both Inbound and outbound voice traffic and can integrate with PSTN or any PBX over E1 or SIP protocol while communicating with the conversational AI application over WebSockets.

BotStream comprises one or more telephony switches, front ended by a SIP proxy for load balancing. This allows BotStream to scale horizontally as the load increases. The BotStream is built on Epicode's IraCluster platform.

1.1 Primary Opensource Dependencies

1.1.1 NATS

The IraCluster uses NATS for all high-speed inter-process communication. NATS must be installed within the Kubernetes cluster. The high speed messaging between the telephony switches and the VoiceAI engines is enabled via NATS platform.

1.1.2 SQLite

IraCluster uses SQLite library to implement the distributed SQL embedded in-memory transient private database. Any change made to one copy of the db is replicated instantly in every copy of the database.

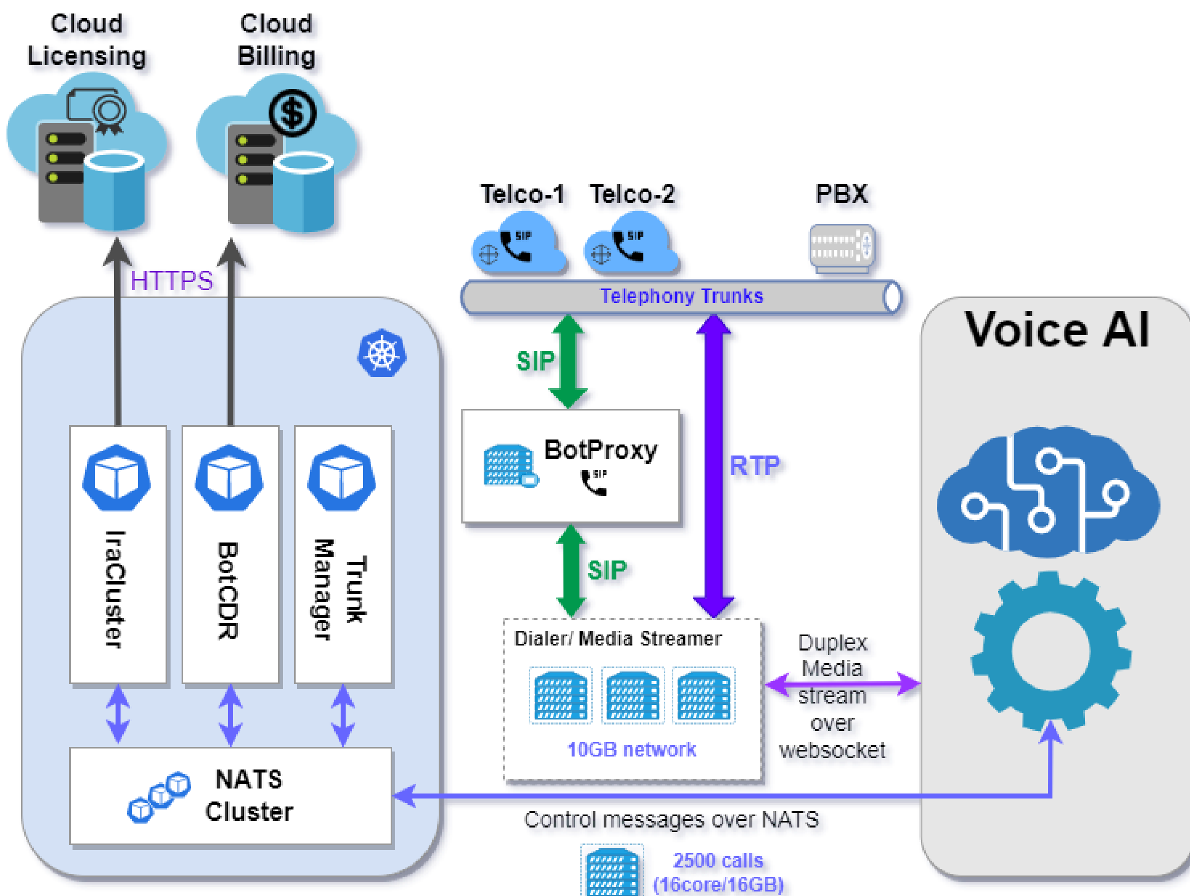
1.1.3 Freeswitch

The SIP and Media is handled using Freeswitch Telephony Platform, which can manage very high call count in each instance. Botstreams uses multiple instances of Freeswitch in a single installation to achieve horizontal scalability.

1.1.4 Kubernetes

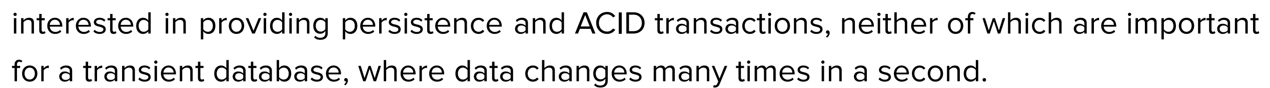
Kubernetes has component level service discovery, redundancy, load balancing and orchestration features, upon which IraCluster implements application layer service discovery, redundancy and load balancing.

2 BotStream Architecture



The VoiceAI engines communicate with BotStream in two different ways. NATS for sending requests and receiving events. Web Socket for sending and receiving raw audio data. The BotStream deals with all the VOIP communication with telephony trunk providers.

This architecture has no theoretical limits when it comes to scalability. It can be scaled both vertically and horizontally depending on the requirements. This is made possible by employing a combination of mature technologies. The biggest challenge to scalability is the state synchronization across multiple instances of applications that must behave like a swarm, a group with a single mind. None of the traditional databases or in-memory databases provide the speed necessary to synchronize state, since they are more



interested in providing persistence and ACID transactions, neither of which are important for a transient database, where data changes many times in a second.

All interactions with BotStream, IraCluster and IraTDB will be through NATS messaging.

2.1 IraCluster

As the reader may be aware, Kubernetes cluster is a collection of containers that are managed or orchestrated by the Kubernetes layer. However, there is a need for a higher level of orchestration for stateful applications to communicate with each other. IraCluster is one such layer to seamlessly integrate multiple applications to work together towards a common goal while providing redundancy and high availability.

IraCluster provides:

1. Tracking of all containers that belong to a distributed application instance
2. Provide instant and secure communication between all such containers.
3. Manage floating licensing of all the components that require licensing.
4. Provide a super-fast and powerful distributed in-memory private transient SQL database.
5. IraCluster heavily depends on mature open source products to achieve maximum reusability and high resiliency, and also high acceptability thanks to various connectivity and language options.

Each service within IraCluster must be aware of the existence of other services it needs to interact with. It is achieved via IraCluster service discovery powered by the NATS platform. One can run multiple IraClusters within a single Kubernetes cluster by using different IraCluster names. The IraCluster is made possible by a bunch of federated services running on independent containers. Even components running outside the kubernetes cluster can join IraCluster using NATS.

2.2 IraTDB

A global variable is an extremely convenient way to store, access, manipulate transient data. It is very fast, very private and data goes away when the application shuts down. Same concept can be extended to a full fledged relational database when in-memory SQLite is used within an application.

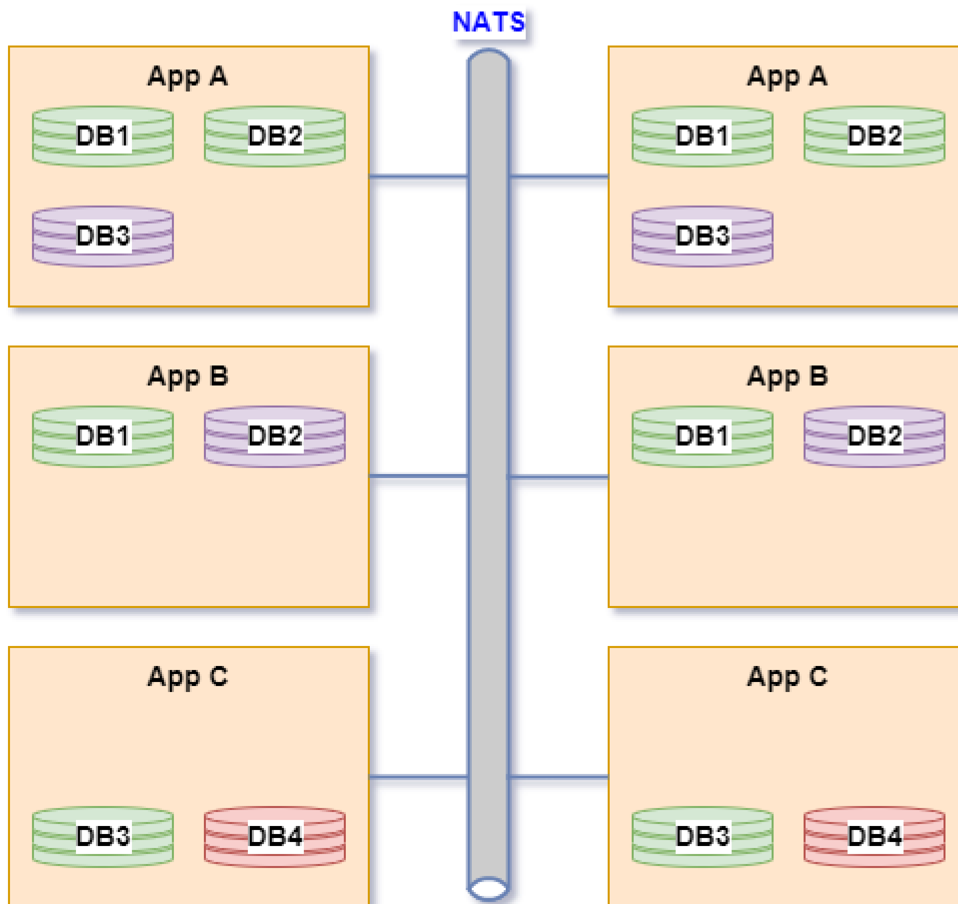
However, in distributed applications or high availability applications, it becomes impossible to retain the same functionality. The storage gets pushed out to an external in-memory database which may be standalone or distributed. It won't be as fast as an embedded database where data resides in heap memory. And it won't be private since other applications can access the data provided they have database admin privileges. It is impossible to keep the data inaccessible/immutable outside of the application while using any external database.

IraTDB is a transient real time embedded database which is a global variable with SQL query engine, which is also distributed. It is secured at database level by RSA keys. Even read and write is separately controlled by keys. Since IraTDB is embedded into the application storing only the database application uses, it is also serverless.

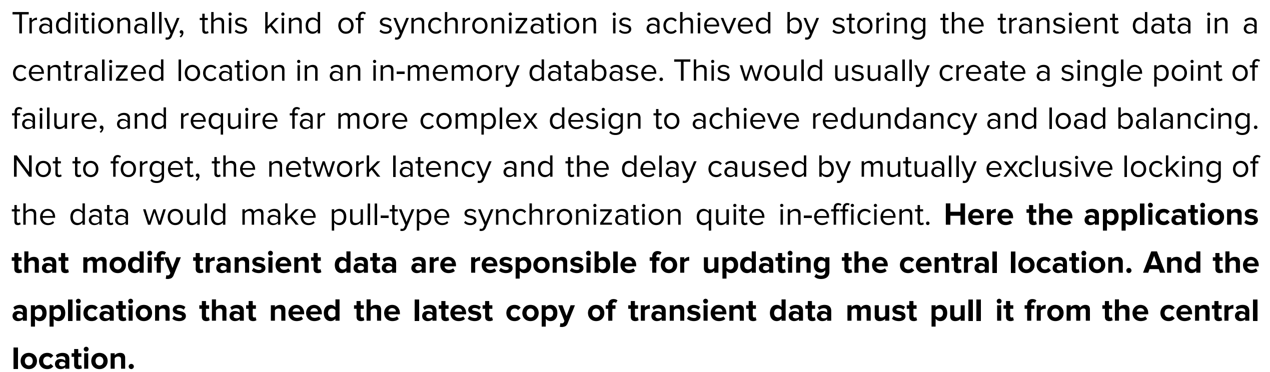
Applications that can't directly link or embed the C++ implementation, can communicate with iratdbot which is a standalone application that accepts NATS commands to open and operate IraTDB. IraTDB uses SQL for all the database operations.

2.2.1 IraTDB Architecture

IraTDB makes use of the fact that one can create multiple in-memory SQLite databases in each application. Each application can have full, read-only and none permission on various databases. The permission is decided by whether the consuming application has the keys to unlock the replicated data.



The databases will replicate into every copy of the applications that subscribe to those databases. For example, in the above figure DB1 will exist in every copy of App A and App B, and DB3 will exist in App A and App C, and DB4 will exist only in copies of App C. Each database in the figure is a SQLite memory database. Green means read/write, violet means read-only and red means private to that application copies. Other applications can't get it even if they subscribe to it.



Traditionally, this kind of synchronization is achieved by storing the transient data in a centralized location in an in-memory database. This would usually create a single point of failure, and require far more complex design to achieve redundancy and load balancing. Not to forget, the network latency and the delay caused by mutually exclusive locking of the data would make pull-type synchronization quite in-efficient. **Here the applications that modify transient data are responsible for updating the central location. And the applications that need the latest copy of transient data must pull it from the central location.**

To solve this problem, a different approach was considered using the IraCluster framework. An application can create a named database within the process, and changes to it are published to a NATS subject. Any application that opens that named database will subscribe to that subject. This database is essentially located in the heap memory of the process. When any db table is modified (insert/update/delete), the changes will also be replicated asynchronously to every instance of all the applications in IraCluster that are subscribed to that database. These replications are handled by the IraTDB layer, and are invisible to the process. **Here the applications that modify transient data are responsible for updating only local data within the process. The applications that need the latest copy of transient data will always find it within the process.** Contrast this with the scenario in the previous paragraph. Since the processes always work with the copy they already have within, the performance improvement is phenomenal.

3 Scalability using Application Load Balancing

Uniform load distribution across all the instances is the key to achieving scalability in distributed telephony applications. We have been able to achieve 2500 simultaneous calls in a single 16 core server, and multiple such instances can exist in a single cluster, as long as network bandwidth is sufficient. We recommend a 10Gbps network or more.

3.1 Outbound Traffic

In distributed dialers, load balancing cannot be achieved by using a traditional network balancer. That is because the load in a dialer system is outcome based. If you dial out 100 calls by sending dial requests via a network balancer sitting over 4 instances of BotStream, it is possible to dial 25 calls from each BotStream. However, only 30 of the

calls may connect to a customer, and those connected calls won't be uniformly distributed across 4 instances. The distribution will be skewed and simply out of control, some taking on too much load, and some running idle.

By using the IraTDB powered by IraCluster, multiple instances of BotStream are always aware of how many calls are running in each instance. Instead of using network load balancer, the BotStream uses NATS queuing to accept the request in any one of the instances. After that the request is forwarded to the BotStream with the least load. This ensures that every BotStream has almost the same amount of load. New BotStream instances can be added during production and they will get into action instantly, taking on all the new requests until the load matches the older servers. Similarly, BotStreams can be taken out by marking one of them to be inactive until all the ongoing calls are completed, after which it can be shut down.

3.2 Inbound Traffic

The inbound calls are SIP/RTP traffic, which uses a stateful protocol. A network load balancer is again useless. The installation must have the ability to distribute while considering the actual load on each BotStream instance. Epicode's BotProxy has instant access to the load data, and can direct the incoming calls to the instance with least traffic thus achieving uniform load distribution.

There is no need to use a network load balancer in BotStream installations.

4 Conclusion

BotStream, Epicode's Flagship product under the category " Conversational AI Enablers" has been specifically designed and developed to address the unique requirements of VoiceAI applications. The competing products in the market are very generic in nature and they do not provide the flexibility that VoiceAI application partners expect. For example, BotStream supports an unlimited licensing model, wherein, partners need not have to request for additional licenses whenever they need to increase the traffic load thereby avoiding bureaucratic delays; resulting in revenue loss. BotStream allows partners to launch any number of additional VoiceBots as long as they have the CPU resources and SIP trunk channels to handle the additional load.