|  |  |  |  |
| --- | --- | --- | --- |
| Document Control | | | |
| Version | Date | Editor | Comments |
| 0.1 | 09/06/2020 | Braun Brelin | Initial document draft |
| 0.2 | 14/06/2020 | Braun Brelin | Revise Publish/Subscriber section as per feedback |

Diffusion Cloud User Manual

09/06/2020

# Introduction

Diffusion cloud from Push Technologies is intelligent cloud-based software that makes it easy for programmers to develop applications that need to run in real-time and retrieve and process data from many different remote sources.

These applications span the range from Web based, Mobile, Internet of Things (IOT) and any other applications that uses distributed data.

# The benefits of Diffusion cloud.

## Robust and Reliable.

<image goes here >

Diffusion cloud will allow you to manage your data even over slow or unreliable networks.

When a connection is dropped, Diffusion will automatically remember its current state and reconnect and resume processing when the connection is re-established. Diffusion will guarantee reliability of data transmission, even in the worst case network scenarios.

## Secure.

<image goes here>

Diffusion cloud uses role based security to ensure that only authorized clients are able to access your data. In addition, Diffusion employs different encryption algorithms to ensure that malicious third parties can’t snoop on and steal your data when you’re sending or receiving it over the network.

## Platform Independent.



Diffusion cloud supports all standard operating systems, including Microsoft Windows, Linux and MacOS X. Additionally, we give you client libraries for a variety of different software languages, including Java, JavaScript, C, Microsoft’s .NET framework, Android, IOS and many other platforms.

## Easy to get started and use.

<image goes here>

Push Technologies provides you with easy to read and understand documentation as well as many different coding examples and use cases of our technology.

## Cloud Based.

<image goes here>

Diffusion is supported on many different cloud platforms, including all major commercial platforms such as Amazon AWS, Microsoft Azure and Google Cloud. This will allow you to quickly set up and use our product without having to purchase expensive hardware.

# Use cases for Diffusion cloud

There are numerous applications where Diffusion cloud is the right choice for you.

## Electronic gaming.

E-gaming applications often have fluctuating network loads, with spikes in traffic happening at specific times, such as betting on matches. The ability to continuously deliver service to users in real-time is a critical part of a successful e-gaming system. Additionally, as new offerings are made available, Diffusion cloud can rapidly help you expand your product line easily and quickly.

## Financial services.

Many financial services, such as equity data feeds, must be able to deliver data in real time with little or no lag in service. Large financial institutions as well as smaller shops have a critical need to be able to both produce and consume vast quantities of data quickly. Diffusion cloud provides the ability to rapidly scale your application as your network and data requirements increase.

## Transportation and Logistics.

The need to both travel and ship goods domestically and internationally means that software applications for logistics and travel must be able to rapidly access and report data such as parcel location, flight information, pricing and other data services becomes a critical requirement. Diffusion cloud will give you the power to reliably and robustly deliver the information without huge up front expenditures on hardware.

## Broadcast and Media.

Streaming and broadcast media require large data throughputs and are very sensitive to lags in performance. Diffusion cloud solves this problem by providing the ability to move large volumes of data in both directions and in real time.

# Implementing the Publish/Subscribe model with Diffusion cloud.

## What is the Publish/Subscribe model?

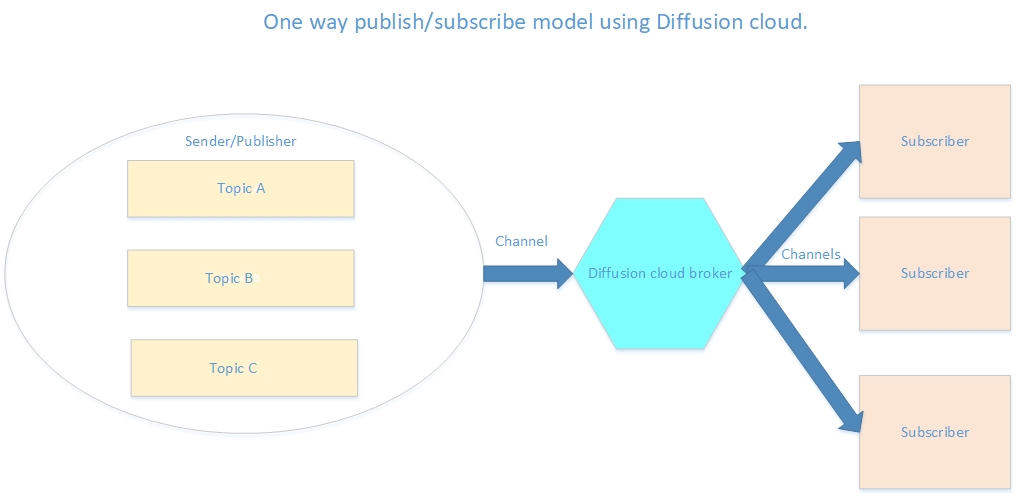
The pub/sub model is a specific pattern that allows *publishers* to send message out to a *subscriber* base. The subscribers request a subscription to a specific set of publishers. When the publisher sends a message its subscriber base, the subscribers receives the specific topics that it has subscribed to and is able to consume and process them.

## Why use the Publish/Subscribe model?

This software model is perfect for cloud based applications. Cloud applications are often decentralized, with different components running on different servers or software containers.

With Pub/Sub, there is no direct coupling between the publishers and the subscribers. Publishers can push data out to subscribers without having to know anything about the subscribers. Subscribers, in turn, can subscribe to any number of publishers without any restrictions, save for any data security constraints.

Figure 1 shows an example of the publisher subscriber model.



Note that there is no direct connection between the publishers and the subscribers. All data flows through the Diffusion cloud broker engine. Because of this, publishers can also be subscribers, and vice versa.

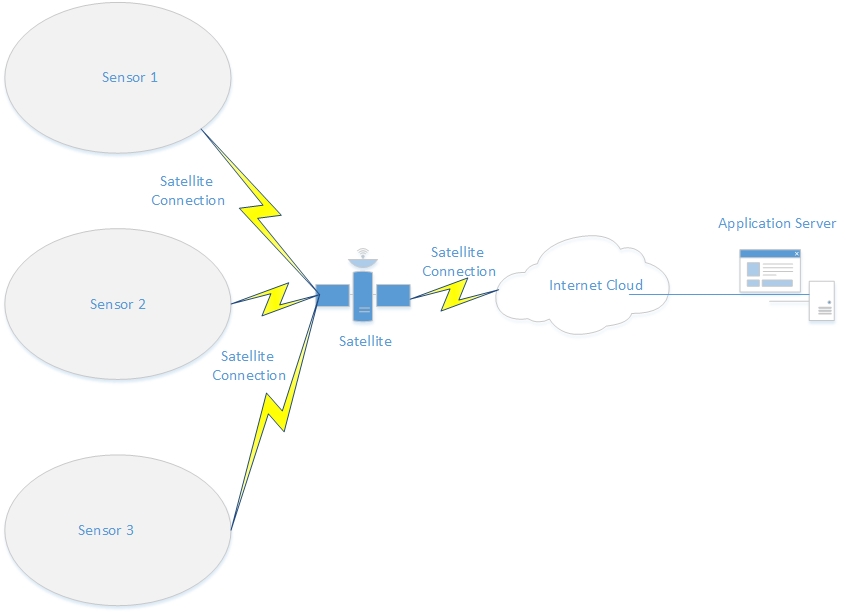
# Using the Publisher/Subscribe model with Diffusion.

Diffusion natively supports programmers that wish to use this model with Diffusion. It supports the creation of logical channels called *topics*. Subscribers subscribe to various topics pushed out by the publisher. Creating a new topic can be as simple as a few lines of code in Java or similar language.

One major difference with Diffusion over other models is that Diffusion pushes data to subscribers rather than waiting for the subscribers to pull the data. This allows Diffusion to only send binary changes to data rather than re-sending data that has already been pushed out. This allows Diffusion to achieve very high network throughput rates.

Consider the following application: You are writing an application that monitors remote sensors for the national meterological service. There are three weather sensors that will transmit temperature data on a minute by minute basis over a satellite connection to the back office, where the server that runs your application is stored and executed.

This is an illustration of what the architecture of this service might look like:

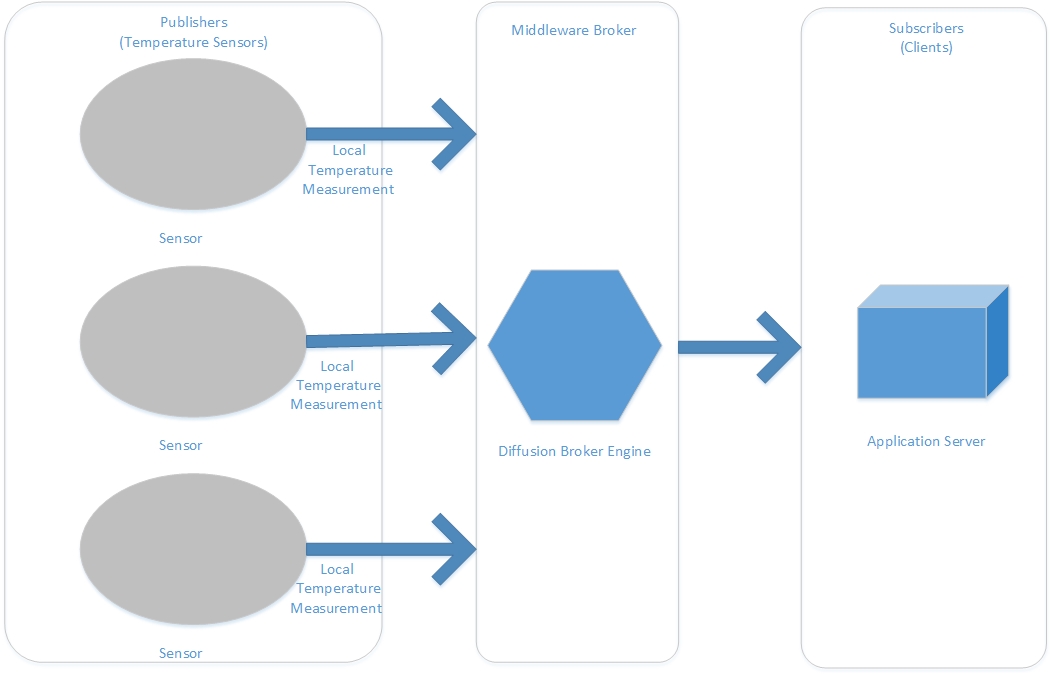


Key challenges in this architecture design include:

1. Low and potentially unreliable network transmission.
2. Potentially high network load as more and more sensors are added to the network.

In each case, Diffusion provides a solution to these challenges.

The logical publish/subscription architecture design for this might look as follows:



In this architecture, we have three publishers, the temperature sensors. Each sensor publishes on one topic, called ‘Local Temperature Measurement’. Our application server is the client that has subscribed to each topic and receives data pushed to it by the Diffusion broker.

# Implementing Diffusion Publish/Subscribe concepts.

In the Diffusion publish/subscribe model, there are seven different concepts that can be implemented in software.

* **Update**

An update is data published to a topic by a client that is applied to the topic to change the topic state. The updated data is then pushed out to all subscribing clients.

* **State**

The latest published values of all data items on the topic. The state of a topic is stored on Diffusion Cloud.

* **Value**

A value is an update that contains the current state of all data on the topic.

* **Delta**

A delta is an update that contains only those items of data that have changed on the topic since the last update was sent.

* **Topic loading**

When a client first subscribes to a topic, it is sent a topic load message. A topic load is a value update that contains the current state of the topic.

* **Fetch**

A request for the current state of all data on the topic. A client can fetch a topic's state without being subscribed to the topic. This request-response mechanism of getting data from a topic is separate from topic subscriptions.

* **Topic notifications**

A client can register to receive topic notifications which provide information about which topics exist in the topic tree, but not the topic values. This is useful if your client needs to monitor the structure of the topic tree (or part of the tree) without the overhead of receiving all the values. Registering for notifications is separate from subscribing to a topic.

Let’s see how we can implement these concepts with Java and the Diffusion broker using our sensor network to create a simple publish and subscribe application with the sensors acting as the publishers, and client running on an application server acting as the subscriber.

## Publishing: Topic Creation and Update

We can create topics calling appropriate Diffusion method calls. Note that Diffusion uses the idea of a *future*, which is a form of asynchronous I/O min Java to implement and update topics.

Here we see a quick example of creating and updating a topic.

|  |
| --- |
| public final class PublishingClient {  /\*\*  Main  \*/  public static void main(String… arguments) throws InterrupedException,ExecutionException, TimeoutException {  // Connect using a principal with ‘modify\_topic and update\_topic permissions  final Session session =  Diffusion.sessions().principal(‘principal’.(‘password’).open(‘ws://host:80  // Get the topic control and topic update feature.  final TopicControl topic Control = session.feature(TopicControl.class);  final TopicUpdateControl updateControl = session.feature(TopicUpdateControl.class);  // Create a 64 bit integer topic ‘sensor/temperature’.  final CompletableFuture<TopicControl.AddTopicResult> future =  topicControl.addTopic(  “sensor/temperature”,  “topicControl.newSpecification (TopicType.INT64);    // Wait for the CompletableFuture to to complete.  future.get(10,TimeUnit.SECONDS);  // Update the topic  final UpdateCallBack updateCallback = new UpdateCallback.Default();  for (int i = 0; I < 1000; i++) {  // Use the non-exclusive updater to update the topic without looking at it.  updateControl.updater().valueUpdater(Integer.class).update(  “sensor/temperature”,10,update.Callback);  Thread.sleep(1000);  }  }  } |

In this example, we’re creating a new publisher that will publish a 64 integer temperature value obtained from a sensor. This published value will be assigned to a *topic.* We start by connecting to the Diffusion broker with permission settings allowing the publisher to modify and update the topic.

The underlying mechanism that creates this is called a *future*.To explore more about how futures in Java work, you can click on [this](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Future.html) link.

Next, we update the topic by iterating a thousand times and incrementing the temperature value by one each time, then publishing that value.

Now, we’ll see how to create a simple subscriber with Java and the Diffusion broker.

## Subscribing: Topic Loading and Fetching.

In this next example, we see how to create a simple subscriber to illustrate how to load a topic and fetch its value. As before, we will use Java and the Diffusion broker to implement the client.

|  |
| --- |
| public final class ClientConsumingSimpleTopics {  private static final Logger log = LoggerFactory.getLogger(ClientConsumingSimpleTopics.class);  private final Session session;  // Constructor  public ClientCosumingSimpleTopics(String serverUrl,  final StringListener stringListener) {  session = Diffusion.sessions().principal(‘client’).password(‘password’).  Open(serverUrl);  // Use the Topics feature to add stream and subscribe to the topic.  final Topics topics = session.feature(Topics.class);  topics.addStream(“sensor/temperature”,  String.class,  new Topics.ValueStream.Default<String>() {  @Override  Public void onValue(  String topicPath,  TopicSpecification specification,  String oldValue,  String newValue) {  stringListener.onNewValue(topicPath,newValue);  }  });  topics.subscribe(‘sensor/temperature’).  whenComplete((voidResult, exception) -> {  if (exception != null) {  LOG.info(‘subscription failed’, exception);  }  });  }  // Close session  public void close() {  session.close();  }  // Notification of new String value.  public interface StringListener {  void onNewValue(String topic, String value);  }  } |

The above code demonstrates the concepts in Diffusion of accessing a topics state and value as well as performing topic loading and fetching.

In the above example, we see the client connecting to the broker, loading the topic that it wants to subscribe to via the broker, and then waiting for the topics value to change, at which time it fetches the new value that it can use for further processing.

Note that in this simple example with a single 64 bit integer value, that value will always be sent to the subscribers since it changes constantly, however, if the topic contained a second value, and that value did not change, then Diffusion would not send this unchanged value over the network as it does not provide any benefit and consumes network bandwidth unnecessarily.