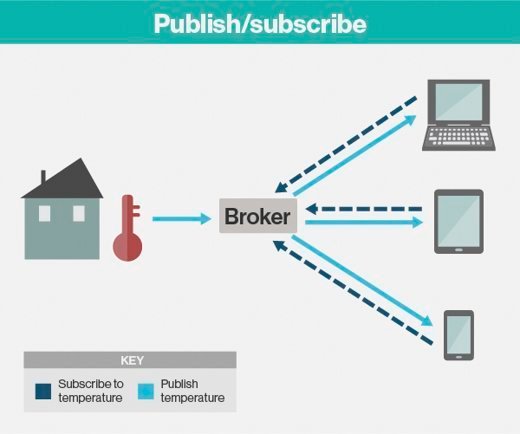
**MQTT**

**What is MQTT ?**

MQTT (MQ Telemetry Transport) is a lightweight messaging protocol that provides resource-constrained network clients with a simple way to distribute [telemetry](https://whatis.techtarget.com/definition/telemetry)information. The protocol, which uses a publish/subscribe communication pattern, is used for machine-to-machine ([M2M](https://internetofthingsagenda.techtarget.com/definition/machine-to-machine-M2M)) communication and plays an important role in the internet of things ([IoT](https://internetofthingsagenda.techtarget.com/definition/Internet-of-Things-IoT)).

MQTT enables resource-constrained IoT devices to send, or publish, information about a given topic to a server that functions as an MQTT [message broker](https://whatis.techtarget.com/definition/message-broker). The broker then [pushes](https://whatis.techtarget.com/definition/push-or-server-push) the information out to those clients that have previously subscribed to the client's topic. To a human, a topic looks like a hierarchical file path. Clients can subscribe to a specific level of a topic's hierarchy or use a [wild-card character](https://whatis.techtarget.com/definition/wildcard-character) to subscribe to multiple levels.

The MQTT protocol is a good choice for wireless networks that experience varying levels of [latency](https://whatis.techtarget.com/definition/latency) due to occasional [bandwidth](https://searchnetworking.techtarget.com/definition/bandwidth) constraints or unreliable connections. Should the connection from a subscribing client to a broker get broken, the broker will buffer messages and push them out to the subscriber when it is back online. Should the connection from the publishing client to the broker be disconnected without notice, the broker can close the connection and send subscribers a cached message with instructions from the publisher.



**MQTT's publish/subscribe model**

**How MQTT works**

An MQTT session is divided into four stages: connection, authentication, communication and termination. A client starts by creating a TCP/IP connection to the broker by using either a standard [port](https://searchnetworking.techtarget.com/definition/port-number) or a custom port defined by the broker's operators. When creating the connection, it is important to recognize that the server might continue an old session if it is provided with a reused client identity.

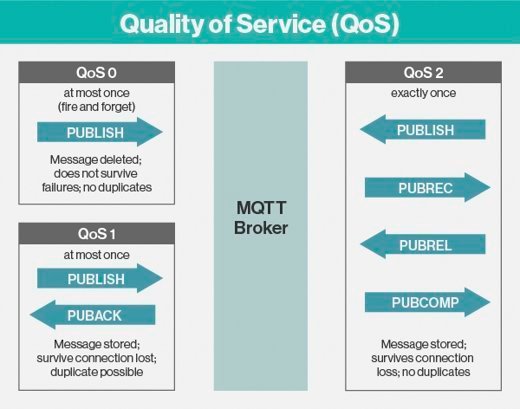
The standard ports are 1883 for non-encrypted communication and 8883 for encrypted communication using [SSL](https://searchsecurity.techtarget.com/definition/Secure-Sockets-Layer-SSL)/[TLS](https://searchsecurity.techtarget.com/definition/Transport-Layer-Security-TLS). During the SSL/TLS handshake, the client validates the [server certificate](https://searchsecurity.techtarget.com/definition/SSL-certificate-Secure-Sockets-Layer-certificate) to authenticate the server. The client may also provide a client certificate to the broker during the handshake, which the broker can use to authenticate the client. While not specifically part of the MQTT specification, it has become customary for brokers to support client authentication with SSL/TLS client-side certificates.

Because the MQTT protocol aims to be a protocol for resource-constrained and IoT devices, SSL/TLS might not always be an option and, in some cases, might not be desired. In such cases, authentication is presented as a clear-text username and password that is sent by the client to the server as part of the CONNECT/CONNACK packet sequence. Some brokers, especially open brokers published on the internet, will accept anonymous clients. In such cases, the username and password are simply left blank.

MQTT is called a lightweight protocol because all its messages have a small code footprint. Each message consists of a fixed header -- 2 [bytes](https://searchstorage.techtarget.com/definition/byte) -- an optional variable header, a message payload that is limited to 256 MB of information and a quality of service ([QoS](https://searchunifiedcommunications.techtarget.com/definition/QoS-Quality-of-Service)) level.

The three different quality of service levels determine how the content is managed by the MQTT protocol. Although higher levels of QoS are more reliable, they have more latency and bandwidth requirements, so subscribing clients can specify the highest QoS level they would like to receive.

The simplest QoS level is unacknowledged service. This QoS level uses a PUBLISH packet sequence; the publisher sends a message to the broker one time and the broker passes the message to subscribers one time. There is no mechanism in place to make sure the message has been received correctly, and the broker does not save the message. This QoS level may also be referred to as at most once, QoS0, or fire and forget.



**MQTT quality of service levels**

The second QoS level is acknowledged service. This QoS level uses a PUBLISH/PUBACK packet sequence between the publisher and its broker, as well as between the broker and subscribers. An acknowledgement packet verifies that content has been received and a retry mechanism will send the original content again if an acknowledgement is not received in a timely manner. This may result in the subscriber receiving multiple copies of the same message. This QoS level may also be referred to as at least once or QoS1.

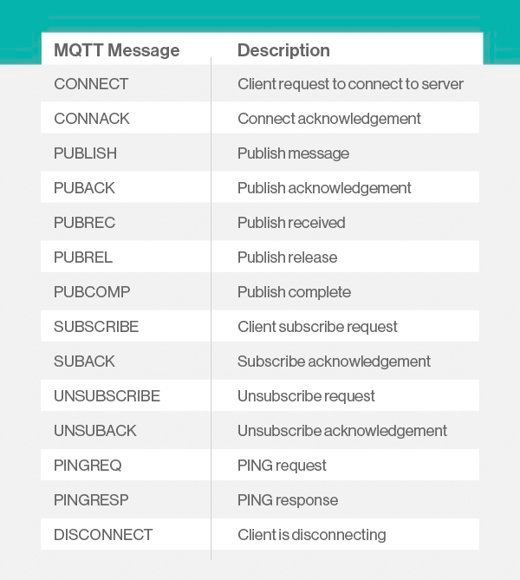
The third QoS level is assured service. This QoS level delivers the message with two pairs of packets. The first pair is called PUBLISH/PUBREC, and the second pair is called PUBREL/PUBCOMP. The two pairs ensure that, regardless of the number of retries, the message will only be delivered once. This QoS level may also be referred to as exactly once or QoS2.

During the communication phase, a client can perform publish, subscribe, unsubscribe and ping operations. The publish operation sends a [binary block](https://searchsqlserver.techtarget.com/definition/block) of data -- the content -- to a topic that is defined by the publisher.

MQTT supports message [BLOBS](https://searchsqlserver.techtarget.com/definition/BLOB) up to 256 MB in size. The format of the content is application-specific. Topic subscriptions are made using a SUBSCRIBE/SUBACK packet pair. Unsubscription is similarly performed using an UNSUBSCRIBE/UNSUBACK packet pair.

Topic strings form a natural topic tree with the use of a special delimiter character, the forward slash (/). A client can subscribe to -- and unsubscribe from -- entire branches in the topic tree with the use of special wild-card characters.

There are two wild-card characters: a single-level wild-card character, the plus character (+); and a multilevel wild-card character, the hash character (#). A special topic character, the dollar character ($), excludes a topic from any root wild-card subscriptions. Typically, the $ is used to transport server-specific or system messages.



**MQTT protocol messages**

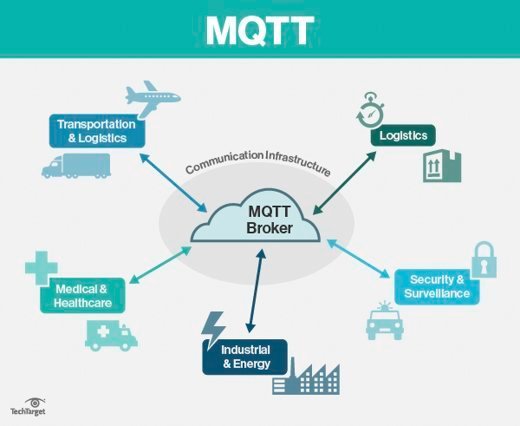
The fourth operation a client can perform during the communication phase is to [ping](https://searchnetworking.techtarget.com/definition/ping) the broker server using a PINGREQ/PINGRESP packet sequence, which roughly translates to ARE YOU ALIVE/YES I AM ALIVE. This operation has no other function than to maintain a live connection and ensure the TCP connection has not been shut down by a [gateway](https://internetofthingsagenda.techtarget.com/definition/gateway) or router.

When a publisher or subscriber wants to terminate an MQTT session, it sends a DISCONNECT message to the broker, and then closes the connection. This is called a graceful shutdown because it gives the client the ability to easily reconnect by providing its client identity and resuming where it left off.

Should the disconnect happen suddenly without time for a publisher to send a DISCONNECT message, the broker may send subscribers a message from the publisher that the broker has previously cached. The message, which is called a last will and testament, provides subscribers with instructions for what to do if the publisher dies unexpectedly.

**MQTT protocol applications and use cases**

Facebook currently uses MQTT for their messenger app, not only because the protocol conserves battery power during mobile phone-to-phone messaging, but also because, in spite of inconsistent internet connections across the globe, the protocol enables messages to be delivered efficiently in milliseconds.



**MQTT vertical applications**

Most major cloud services providers, including AWS, Google Cloud, IBM Bluemix and Microsoft Azure, support MQTT, as do the Carriots, Evrything and ThingWorx IoT platforms.

MQTT is well-suited to applications using M2M and IoT devices for real-time analytics, preventative maintenance and monitoring, among other uses, in environments such as smart homes, healthcare, logistics, industry and manufacturing.

**Competing protocols**

Other transfer protocols under consideration for IoT devices with constrained resources include the Constrained Application Protocol (CoAP), which uses a request/response communication pattern, and the Advanced Message Queuing Protocol (AMQP), which, like MQTT, uses a publish/subscribe communication pattern.

**Connection Strings**

There're three types of connection strings in Azure IoT Hub:

* IoT Hub connection string
* IoT Hub's Event Hub-compatible connection string
* IoT Hub device connection string

### IoT Hub Connection String

Used mainly for device registration/un-registration.

**Value format**

HostName=<Host Name>;SharedAccessKeyName=<Key Name>;SharedAccessKey=<SAS Key>

### IoT Hub's Event Hub-compatible connection string

**Usage**

IoT Hub messages are saved in built-in endpoint Events by default, you may use this endpoint as event hub with the event hub-compatible connection string

**Value** **format**

Endpoint=<ENDPOINT>;SharedAccessKeyName=<Key Name>;SharedAccessKey=<KEYVALUE>  
Please note <ENDPOINT> starts with sb://

The value is similar as IoT Hub connection string except the first part is endpoint instead of host name.

### IoT Hub device connection string

**Usage**

Mainly used by devices to

* + - Send Device to Cloud messages
    - Receive Cloud to Device messages
    - Response direct method

**Value** **format**

HostName=<Host Name>;DeviceId=<Device Name>;SharedAccessKey=<Device Key>

## **Azure Event Hubs and Azure Stream Analytics**

## **What is Azure Event Hubs?**

Azure Event Hubs is a Big Data streaming platform and event ingestion service, capable of receiving and processing millions of events per second. Event Hubs can process and store events, data, or telemetry produced by distributed software and devices. Data sent to an event hub can be transformed and stored using any real-time analytics provider or batching/storage adapters.

Event Hubs is used in some of the following common scenarios:

* Anomaly detection (fraud/outliers)
* Application logging
* Analytics pipelines, such as clickstreams
* Live dashboarding
* Archiving data
* Transaction processing
* User telemetry processing
* Device telemetry streaming

## **Why use Event Hubs?**

Data is valuable only when there is an easy way to process and get timely insights from data sources. Event Hubs provides a distributed stream processing platform with low latency and seamless integration, with data and analytics services inside and outside Azure to build a complete Big Data pipeline.

Event Hubs represents the "front door" for an event pipeline, often called an *event ingestor* in solution architectures. An event ingestor is a component or service that sits between event publishers and event consumers to decouple the production of an event stream from the consumption of those events. Event Hubs provides a unified streaming platform with time retention buffer, decoupling the event producers from event consumers.

## **Key features**

Event Hubs provides message stream handling capability but has characteristics that are different from traditional enterprise messaging. Event Hubs capabilities are built around high throughput and event processing scenarios. Event Hubs contains the following [key components](https://docs.microsoft.com/en-us/azure/event-hubs/event-hubs-features):

**Event producers**: Any entity that sends data to an event hub. Event publishers can publish events using HTTPS or AMQP 1.0 or Apache Kafka (1.0 and above)

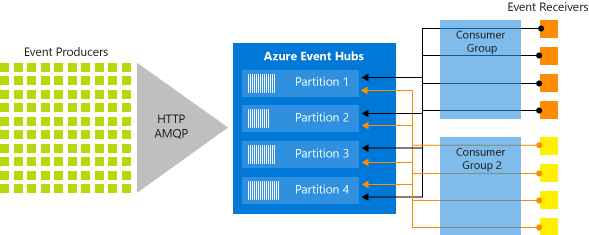
**Partitions**: Each consumer only reads a specific subset, or partition, of the message stream.

**Consumer groups**: A view (state, position, or offset) of an entire event hub. Consumer groups enable multiple consuming applications to each have a separate view of the event stream, and to read the stream independently at their own pace and with their own offsets.

**Throughput units**: Pre-purchased units of capacity that control the throughput capacity of Event Hubs.

**Event receivers**: Any entity that reads event data from an event hub. All Event Hubs consumers connect via the AMQP 1.0 session, and events are delivered through the session as they become available.

## The following figure shows the Event Hubs stream processing architecture:



## **What is Microsoft Azure Stream Analytics ?**

Microsoft Azure Stream Analytics is a serverless scalable complex event processing engine by [Microsoft](http://www.wiki-zero.co/index.php?q=aHR0cHM6Ly9lbi53aWtpcGVkaWEub3JnL3dpa2kvTWljcm9zb2Z0) that enables users to develop and run real-time analytics on multiple streams of data from sources such as devices, sensors, web sites, social media, and other applications. Users can set up alerts to detect anomalies, predict trends, trigger necessary workflows when certain conditions are observed, and make data available to other downstream applications and services for presentation, archiving, or further analysis.

**Azure Event Hubs and Stream Analytics working together**

Azure Event Hubs provides highly scalable publish-subscribe event ingestors. An event hub can collect millions of events per second, so that you can process and analyze the massive amounts of data produced by your connected devices and applications. Together, Event Hubs and Stream Analytics provide an end-to-end solution for real-time analytics. Event Hubs lets you feed events into Azure in real-time, and Stream Analytics jobs can process those events in real-time. For example, you can send web clicks, sensor readings, or online log events to Event Hubs. You can then create Stream Analytics jobs to use Event Hubs as the input data streams for real-time filtering, aggregating, and correlation.

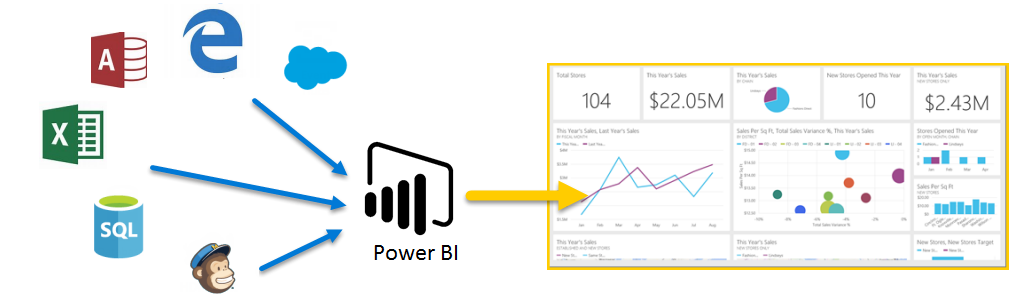
**Why we use consumer groups ?**

You should configure each Stream Analytics event hub input to have its own consumer group. When a job contains a self-join or has multiple inputs, some inputs might be read by more than one reader downstream. This situation impacts the number of readers in a single consumer group. To avoid exceeding the Event Hubs limit of five readers per consumer group per partition, it's a best practice to designate a consumer group for each Stream Analytics job. There is also a limit of 20 consumer groups per event hub.

**Microsoft Power BI**

**What is Microsoft Power BI ?**

Power BI is a collection of software services, apps, and connectors that work together to turn your unrelated sources of data into coherent, visually immersive, and interactive insights. Whether your data is a simple Excel spreadsheet, or a collection of cloud-based and on-premises hybrid data warehouses, Power BI lets you easily connect to your data sources, visualize (or discover) what’s important, and share that with anyone or everyone you want.



Power BI can be simple and fast – capable of creating quick insights from an Excel spreadsheet or a local database. But Power BI is also robust and enterprise-grade, ready for extensive modeling and real-time analytics, as well as custom development. So it can be your personal report and visualization tool, and can also serve as the analytics and decision engine behind group projects, divisions, or entire corporations.