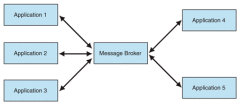
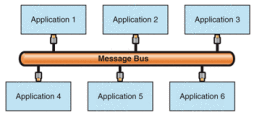
Message Broker or Bus – what’s the difference?

In the process of evaluating and trialling the introduction of messaging to integrate distributed back-end services, and other apps, in our system, the question of whether we were aiming to design a Message Broker or a Message Bus arose. This led to me doing some research to verify and improve my understanding of the difference. This post is a summary of my findings.

در فرایند ارزیابی و ترسیم کردن معرفی پیام رسانی برای ادغام سرویس های توزیع شده پشتیبان و سایر برنامه ها ، در سیستم ما این سؤال وجود دارد که آیا ما قصد داشتیم یک کارگزار پیام یا یک اتوبوس پیام طراحی کنیم. این امر باعث شد که من برخی از تحقیقات را برای تأیید و بهبود درک من از تفاوت انجام دهم. این پست خلاصه ای از یافته های من است

[](https://neiljbrown.files.wordpress.com/2017/05/message-broker-mediating-collaboration.png)[](https://neiljbrown.files.wordpress.com/2017/05/message-bus-toplogy.png)

**Messaging Topologies**

Firstly, let’s be clear, the terms Message Broker and Message Bus are used in architectural patterns for messaging systems, also referred to as messaging topologies. Whilst a Message Bus is one such topology, a Message Broker is only one component in an alternative topology known as Hub and Spoke. These topologies describe different ways for integrating apps using messaging.

توپولوژی های پیام رسانی

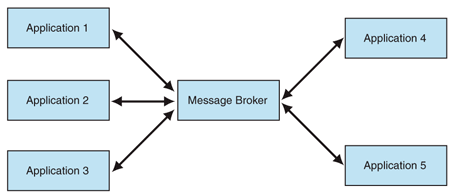
در مرحله اول ، واضح است ، اصطلاحات کارگذار پیام و اتوبوس پیام در الگوهای معماری برای سیستم های پیام رسانی استفاده می شود ، همچنین به عنوان توپولوژی پیام رسانی خوانده می شوند. در حالی که یک اتوبوس پیام یکی از این توپولوژی هاست ، یک کارگزار پیام تنها یک جزء در یک توپولوژی جایگزین شناخته شده به عنوان Hub and Spoke است. این توپولوژی ها روش های مختلفی را برای ادغام برنامه ها با استفاده از پیام رسانی توصیف می کنند.

**Hub and Spoke**

In the Hub and Spoke topology, applications are integrated via a central messaging middleware component. Apps send and receive messages to and from the ‘hub’ only. Senders have no awareness of other apps, or their specific concerns. The hub actively mediates, or ‘brokers’, all communications between applications, to provide additional integration and messaging services. The hub is therefore commonly referred to as a Message Broker.

Hub and Spoke (هاب و صحبت)

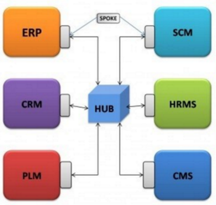
در توپولوژی Hub and Spoke ، برنامه ها از طریق یک مؤلفه پیام رسانی مرکزی ادغام می شوند. برنامه ها فقط پیام ها را به "hub" ارسال و دریافت می کنند. فرستنده ها هیچ آگاهی از برنامه های دیگر یا نگرانی های خاص آنها ندارند. این مرکز بطور فعال واسطه یا "کارگزاران" در کلیه ارتباطات بین برنامه ها ، برای ارائه خدمات یکپارچه سازی و پیام رسانی اضافی است. از این رو hub معمولاً به عنوان یک کارگزار پیام گفته می شود.

[](https://neiljbrown.files.wordpress.com/2017/05/message-broker-mediating-collaboration.png)

Ref [1]

Each app is connected to the hub via a ‘spoke’. These are application-specific protocol and API messaging adapters that are logically part of the hub, rather than the apps. (The spokes are implemented and deployed as extensions of the hub rather than apps).

هر برنامه از طریق "صحبت" به hub متصل می شود. اینها آداپتورهای مخصوص پروتکل و پیام های API مخصوص برنامه ها هستند که به جای برنامه ها ، منطقاً بخشی از hub هستند. (سخنگوها به جای برنامه کاربردی به صورت extensionهایی از hub اجرا و مستقر می شوند)

[](https://neiljbrown.files.wordpress.com/2017/05/hub-and-spoke-topology.png)

The hub aims to fully decouple apps from one another, such that a sending application has no awareness of either the number or destination / location of consuming apps.

هدف hub جدا کردن کامل برنامه ها از یکدیگر است به گونه ای که یک برنامه ارسال کننده هیچ آگاهی از شماره یا مقصد / محل مصرف برنامه ها ندارد

**Message Bus**

A Message Bus is an architectural pattern that supports integrating apps using a shared (common) set of interfaces. It consists of the following key elements:

* A shared infrastructure (network and messaging channels) for sending messages to recipients.
* A set of agreed-upon message schemas (message headers, and a common data-model in terms of resources and representations used in message payloads).
* A set of common command messages. (Used to support the messaging equivalent of RPC to invoke methods in other apps).

When an app wants to send a message it passes it to the message bus, which is responsible for delivering the message to all the other apps listening for messages on the bus’ shared infrastructure.

اتوبوس پیام

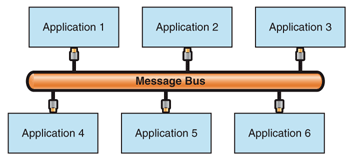
اتوبوس پیام یک الگوی معماری است که از یکپارچه سازی برنامه ها با استفاده از یک رابط مشترک (عمومی) پشتیبانی می کند. این شامل عناصر اصلی زیر است:

• یک زیرساخت مشترک (کانالهای شبکه و پیام رسانی) برای ارسال پیام به گیرندگان.

• مجموعه ای از طرح های پیام توافق شده (هدر پیام ها ، و یک مدل داده مشترک از نظر منابع و بازنمایی های مورد استفاده در payloadهای پیام).

• مجموعه ای از پیام های دستوری عمومی. (برای پشتیبانی از معادل پیام رسانی RPC برای فراخوانی روشها در برنامه های دیگر استفاده می شود).

هنگامی که یک برنامه می خواهد پیامی را ارسال کند ، آن را به اتوبوس پیام منتقل می کند ، که وظیفه دارد پیام را به تمام برنامه های دیگر تحویل دهد که به زیرساخت اشتراکی اتوبوس برای پیامها گوش می دهند.

[](https://neiljbrown.files.wordpress.com/2017/05/message-bus-toplogy.png)

Ref [2]

It’s the apps that are responsible for implementing the common bus Interface. An app that sends messages through the bus must prepare the message so it complies with the type of messages the rest of the apps connected to the bus expect. Similarly, an application that receives messages must be able to understand (syntactically, although not necessarily semantically) the message types.

این برنامه هایی هستند که وظیفه اجرای رابط عمومی اتوبوس را بر عهده دارند. برنامه ای که پیام را از طریق اتوبوس ارسال می کند ، باید پیام را آماده کند تا با نوع پیام هایی مطابقت داشته باشد که بقیه برنامه های متصل به اتوبوس انتظار دارند. به طور مشابه ، برنامه ای که پیام دریافت می کند باید انواع پیام ها را بفهمد (به صورت نحوی ، گرچه نه لزوماً از نظر معنایی)

**Similarities and Differences**

Diagrams like those above visually distinguish the two topologies. Whilst helpful, they can lead to overlooking the fact that messaging systems that are based on both topologies have common features. Whilst both topologies have different strengths (benefits) and weaknesses (liabilities), the primary benefit of both topologies over a point-to-point (direct) integration style is that they reduce the number of integrations that need to be built to fully integrate all the apps in a system – avoiding “integration spaghetti”.

مشابهت ها و تفاوت ها

نمودارهایی مانند موارد فوق بطور بصری دو توپولوژی را از هم متمایز می کند. در عین حال مفید ، آنها می توانند به این واقعیت منجر شوند که سیستم های پیام رسانی که بر اساس هر دو توپولوژی بنا شده اند ، دارای ویژگی های مشترکی هستند. در حالی که هر دو توپولوژی نقاط قوت (مزایا) و ضعف (بدهی) های مختلفی دارند ، مهمترین مزیت هر دو توپولوژی در سبک ادغام نقطه به نقطه (مستقیم) این است که آنها تعداد ادغام هایی را که باید ساخته شوند را کاهش می دهند تا کاملاً برنامه ها در یک سیستم ادغام شود - اجتناب از "اسپاگتی ادغام".

Adopting a Message Bus topology implies applications are integrated using a common protocol and they produce and consume messages that adhere to a common, shared schema, including a data-model (set of resources and their representations). A Message Bus provides little or no centralized messaging services. Sent messages might only be delivered (broadcast) to all connected systems. Although pub-sub messaging, another form of message routing, is typically also supported.

اتخاذ توپولوژی اتوبوس پیام حاکی از آن است که برنامه ها با استفاده از یک پروتکل عمومی ادغام می شوند و پیام هایی را که به یک طرح عمومی و مشترک ، از جمله یک مدل داده (مجموعه ای از منابع و بازنمایی آنها) پایبند هستند ، تولید و مصرف می کنند. یک اتوبوس پیام خدمات پیام رسانی کم و غیرمتمرکز ارائه می دهد. پیامهای ارسالی فقط به کلیه سیستمهای متصل ارسال می شوند (پخش می شوند). اگرچه پیام های pub-sub ، شکل دیگری از مسیریابی پیام ، به طور معمول نیز پشتیبانی می شود.

A Hub and Spoke architecture integrates apps without enforcing a common API (message schema) on them, and supports connecting them using disparate protocols. It does so by providing application specific (e.g. protocol) adapters, and an active, central messaging middleware component. This hub, or Message Broker, understands the messages sent to it, knows which app(s) the message needs to be sent, and can transform a message to meet the schema expected by each app. Because it brokers messages sent between apps it can also provide additional messaging services such as message (e.g. content) routing, aggregation and splitting.

یک معماری Hub and Spoke برنامه ها را بدون اجرای یک API عمومی (طرح پیام) روی آنها یکپارچه می کند و از اتصال آنها با استفاده از پروتکل های متفاوت پشتیبانی می کند. این کار را با ارائه آداپتورهای ویژه برنامه (به عنوان مثال پروتکل) و یک جزء میان افزار فعال پیام رسانی مرکزی انجام می دهد. این hub یا کارگذار پیام پیام های ارسالی را میفهمد ، می داند که پیام برای چه برنامه ای ارسال شده است ، و می تواند پیامی را برای تحقق اسکیما مورد نظر هر برنامه تبدیل کند. از آنجا که واسطه پیامهای ارسال شده بین برنامه ها است ، می تواند سرویس پیام رسانی اضافی مانند مسیریابی پیام (به عنوان مثال محتوا) ، تجمع و تقسیم را نیز ارائه دهد

Because a Hub and Spoke architecture provides protocol conversion and data transformation services it supports integrating apps that can’t be easily changed, such as legacy and third-party apps. A Message Bus based integration does not.

از آنجا که یک معماری Hub and Spoke خدمات تبدیل پروتکل و تبدیل داده را ارائه می دهد ، از برنامه های تلفیقی که نمی توان به راحتی تغییر داد ، مانند برنامه های قدیمی و شخص ثالث پشتیبانی می کند. یکپارچه سازی بر پایه اتوبوس پیام این کار را نمیکند.

The Message Broker in a Hub and Spoke architecture contains more logic and maintains more state than a Message Bus. It may therefore be more difficult to scale.

کارگذار پیام در یک معماری Hub and Spoke حاوی منطق بیشتری است و حالت بیشتری را نسبت به اتوبوس پیام حفظ می کند. بنابراین ممکن است برای مقیاس گذاری دشوارتر باشد

**Summary and Conclusions**

A Message Broker is one part of the Hub and Spoke messaging architecture (topology), as distinct from the Message Bus topology. These two topologies are both alternatives to point-to-point (direct) integrations, and primarily aim to reduce the number of integrations that need to be built to fully integrate all the apps in a system – avoiding so-called “integration spaghetti”.

خلاصه و نتیجه گیری

کارگذار پیام یکی از بخش های معماری(توپولوژی) پیام رسان Hub and Spoke است که با توپولوژی اتوبوس پیام متمایز است. این دو توپولوژی هر دو گزینه جایگزینی برای یکپارچه سازی نقطه به نقطه (مستقیم) هستند و در درجه اول با هدف کاهش تعداد ادغام هایی که باید ساخته شوند برای ادغام کامل همه برنامه ها در یک سیستم کاهش می یابد - اجتناب از اصطلاح "اسپاگتی ادغام".

Both topologies have their strengths and weaknesses. One of the main decision points for choosing between them is whether your centralized messaging infrastructure needs to support integrating apps with disparate protocols and / or message schemas, e.g. because the apps can’t be changed to support a common messaging interface.

هر دو توپولوژی نقاط قوت و ضعف خود را دارند. یکی از اصلی ترین نکات تصمیم گیری برای انتخاب بین آنها این است که آیا زیرساخت پیام رسانی متمرکز شما نیاز به پشتیبانی از برنامه های یکپارچه با پروتکل های متفاوت و / یا طرح های پیام دارد ، به عنوان مثال. زیرا نمی توان برنامه ها را برای پشتیبانی از رابط پیام رسانی مشترک تغییر داد.

In practice, messaging middleware will vary in the extent of Message Broker features they support, over and above the more basic features seen in a Message Bus.

در عمل ، میان افزار پیام رسانی به میزان ویژگی های اساسی واسطه ای که از آنها پشتیبانی می کند ، متفاوت و بیشتر از ویژگیهای پایه ای دیگری است که در یک اتوبوس پیام مشاهده می شود

<https://neiljbrown.com/2017/05/13/message-broker-or-bus-whats-the-difference/>

# Message Bus

## Context

You have an integration solution that consists of applications that are provided by different vendors. These applications run on a variety of platforms. Some of these applications generate messages and many other applications consume the messages.

اتوبوس پیام

زمینه

شما یک راه حل ادغام دارید که متشکل از برنامه هایی است که توسط فروشندگان مختلف ارائه می شود. این برنامه ها بر روی سیستم عامل های مختلفی اجرا می شوند. برخی از این برنامه ها پیام ایجاد می کنند و بسیاری از برنامه های دیگر پیام ها را مصرف می کنند.

For example, consider a financial application that integrates trading tools, portfolio management applications, modeling and risk analysis tools, trend indicators, and tickers. Market activity causes interaction between these systems. For example, a trading system communicates the completion of a sell transaction by sending a message to all other trading applications. The trading system could have individual connections to each trading application. This works well for a few applications, but becomes a burden as the number of applications increases. Managing the addition or removal of trading applications should not interfere with processing trades.

به عنوان مثال ، یک برنامه مالی را در نظر بگیرید که ابزارهای معاملاتی ، برنامه های مدیریت سبد سهام ، ابزارهای مدل سازی و تحلیل ریسک ، شاخص های روند و تیک را در اختیار شما قرار دهد. فعالیت بازار باعث تعامل بین این سیستم ها می شود. به عنوان مثال ، یک سیستم معاملاتی با ارسال پیام به کلیه برنامه های تجاری دیگر ، تکمیل معامله فروش را به شما ابلاغ می کند. سیستم تجارت می تواند ارتباطات مجزا با هر یک از برنامه های معاملات داشته باشد. این برای چند برنامه خوب کار می کند ، اما با افزایش تعداد برنامه ها بار سنگین می شود. مدیریت افزودن یا حذف برنامه های تجاری نباید باعث مداخله در معاملات پردازش شود

## Problem

As an integration solution grows, how can you lower the cost of adding or removing applications?

مشکل

با افزایش یک راه حل ادغام ، چگونه می توانید هزینه افزودن یا حذف برنامه های کاربردی را کاهش دهید؟

## Forces

Adding an application to an integration solution or removing an application from an integration solution entails balancing the following forces:

اجبارها

افزودن برنامه به یک راه حل یکپارچه سازی یا حذف برنامه از یک راه حل ادغام مستلزم متعادل کردن نیروهای زیر است:

* Communication between applications usually creates dependencies between the applications. The sender must communicate with the receivers. The receiver must recognize the messages from all the senders. These dependencies translate into coupling between the participants.
* In a configuration where point-to-point connectivity exists, the coupling has a quadratic (or O[n2]) growth with the number of applications [Chandra03, Levine03]. For example, three fully connected applications need three connections, but 10 applications need 45 connections. This quadratic growth hampers maintainability, modifiability, and integrability.
* Usually, the applications of an integration solution have different interfaces. Changing the interfaces of proprietary applications is difficult. Even if you change the interface of one application, it is not feasible to change the interface for all the applications of your integration solution.
* Some integration solutions consist of a fixed set of applications. An integration solution that has low extensibility and modifiability requirements typically does not need to accommodate new applications.
* ارتباط بین برنامه ها معمولاً وابستگی هایی بین برنامه ها ایجاد می کند. فرستنده باید با گیرنده ها ارتباط برقرار کند. گیرنده باید پیام های همه فرستنده ها را تشخیص دهد. این وابستگی ها به جفت شدن بین شرکت کنندگان ترجمه می شود.
* در پیکربندی که اتصال نقطه به نقطه وجود دارد ، اتصال با تعداد برنامه ها [Chandra03 ، Levine03] دارای رشد درجه دوم (یا O [n2]) است. به عنوان مثال ، سه برنامه کاملاً متصل به سه اتصال نیاز دارند ، اما 10 برنامه به 45 اتصال نیاز دارند. این رشد درجه دوم مانع از ماندگاری ، تعدیل و یکپارچگی آن می شود.
* معمولاً کاربردهای یک راه حل ادغام دارای رابطهای متفاوتی هستند. تغییر واسط های برنامه های اختصاصی کاری دشوار است. حتی اگر رابط کاربری یک برنامه را تغییر دهید ، تغییر رابط برای همه برنامه های راه حل یکپارچه سازی امکان پذیر نیست.
* برخی از راه حل های ادغام شامل یک مجموعه ثابت از برنامه ها است. یک راه حل یکپارچه سازی که دارای قابلیت انعطاف پذیری و تعدیل پذیری کم است ، به طور معمول نیازی به برنامه های جدید ندارد.

## Solution

Connect all applications through a logical component known as a message bus. A message bus specializes in transporting messages between applications. A message bus contains three key elements:

راه حل

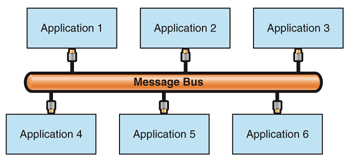
همه برنامه ها را از طریق یک مؤلفه منطقی که به عنوان یک اتوبوس پیام معروف است متصل کنید. اتوبوس پیام در انتقال پیام بین برنامه ها تخصص دارد. اتوبوس پیام شامل سه عنصر اصلی است:

* A set of agreed-upon message schemas
* A set of common command messages [Hohpe04]
* A shared infrastructure for sending bus messages to recipients

When you use a message bus, an application that sends a message no longer has individual connections to all the applications that must receive the message. Instead, the application merely passes the message to the message bus, and the message bus transports the message to all the other applications that are listening for bus messages through a shared infrastructure. Likewise, an application that receives a message no longer obtains it directly from the sender. Instead, it takes the message from the message bus. In effect, the message bus reduces the fan-out of each application from many to one.

Usually, the bus does not preserve message ordering. Internal optimizations, routing, buffering, or even the underlying transport mechanism might affect how the messages travel to the receiving applications. Therefore, the order in which messages reach each receiver is nondeterministic. Preserving the order of messages requires additional logic. This additional logic can be provided by the participating applications. For example, the sending application could insert sequence numbers into the outgoing messages, and the receivers could use those numbers to reorder the incoming messages. The logic could also be provided by the bus, and the logic could therefore be transparent for the participating applications. However, this additional logic is not required.

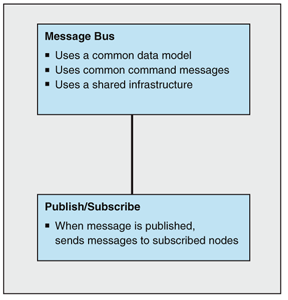
Figure 1 shows an integration solution that uses a message bus. An application that sends messages through the bus must prepare the messages so that the messages comply with the type of messages the bus expects. Similarly, an application that receives messages must be able to understand (syntactically, although not necessarily semantically) the message types. If all applications in the integration solution implement the bus interface, adding applications or removing applications from the bus incurs no changes.



**Figure 1. Applications communicating through a message bus**

The shared infrastructure between a message bus and the listening applications can be achieved by using a Message Router [Hohpe04] or by using a [Publish/Subscribe](https://docs.microsoft.com/en-us/previous-versions/msp-n-p/ff649664%28v%3dpandp.10%29) mechanism. This book focuses on message buses that use Publish/Subscribe mechanisms. For details on how to design a message bus that uses message-oriented middleware and a Message Router, see Enterprise Integration Patterns [Hohpe04].

Figure 2 shows the Message Bus pattern associated with the Publish/Subscribe pattern.



**Figure 2. The Message Bus pattern associated with the Publish/Subscribe pattern**

The kind of Publish/Subscribe implementation that you decide to use with a particular message bus has a profound impact on the message bus architecture. There are three types of Publish/Subscribe implementations: List-Based Publish/Subscribe, Broadcast-Based Publish/Subscribe, and Content-Based Publish/Subscribe.

**Note** Although the Publish/Subscribe pattern is an important part of a message bus, Publish/Subscribe implementations are also used independently of message buses. For example, Publish/Subscribe mechanisms are used with the Point-to-Point Connection and [Message Broker](https://docs.microsoft.com/en-us/previous-versions/msp-n-p/ff648849%28v%3dpandp.10%29) patterns. See the Publish/Subscribe pattern for more details.

### Message Bus with List-Based Publish/Subscribe

Maintaining lists of published topics (subjects) and subscribers (observers) and then notifying each one individually as events occur is the essence of List-Based Publish/Subscribe implementations.

To use a message bus that contains a List-Based Publish/Subscribe implementation, a system sends a command message to the message bus. The message bus then looks up all interested message bus subscribers and sends each message bus subscriber a copy of the original message. Any data that is associated with the message is in a common format so that all systems can interpret the command message and the data, and then respond appropriately.

### Message Bus with Broadcast-Based Publish/Subscribe

To use a Message Bus implementation that contains a Broadcast-Based Publish/Subscribe implementation, a system sends a command message to the message bus. The message bus broadcasts the message to all the nodes that are listening to the bus. The message bus makes no attempt to determine the appropriate subscribers. Instead, each listening node filters any unwanted messages and processes only the messages that it is interested in. Any data that is associated with the message is in a common format so that all systems can interpret the command message and the data, and then respond appropriately.

### Message Bus with Content-Based Publish/Subscribe

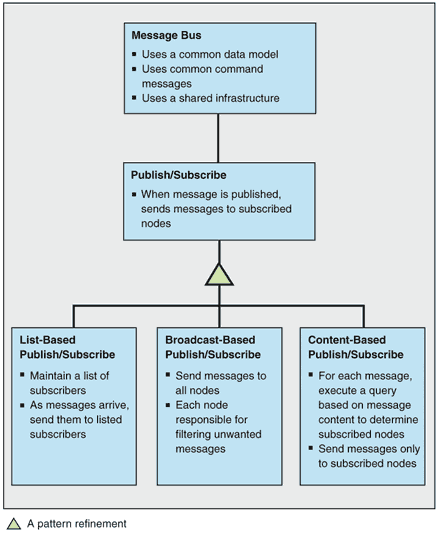
To use a Message Bus pattern that contains a Content-Based Publish/Subscribe implementation, a system sends a command message to the message bus. After the message bus receives the message, it is responsible for matching the message against a set of subscribers and then forwarding the message to each of the appropriate subscribers. To match the message against a set of subscribers, the message bus must determine if there are any subscribers interested in this particular message. If an interested subscriber exists, the subscriber matches a particular message field or fields and a set of values. If a match exists between the message content and a subscriber, the message is then forwarded to each matching subscriber.

After a subscribing system receives a bus message, the subscribing system is able to process the message because the message contains the common command message and the agreed-upon message schemas.

While the Message Bus using List-Based Publish/Subscribe and the Message Bus using Content-Based Publish/Subscribe patterns both check for subscriptions before forwarding messages, there are key differences. The list-based approach matches subscriptions on subjects. However, the content-based approach is much more flexible. Because the content-based approach allows you to identify one or more fields in the message and a set of acceptable values for each field, you can create very intelligent routing capabilities.

To make Content-BasedPublish/Subscribe work, you need a high-performance infrastructure component that can read each message, query a set of potential subscribers, and efficiently route the message to each subscriber. In practice, you can think of this as a message switch. This is exactly how Microsoft BizTalk Server 2004 is designed.

Figure 3 shows the Message Bus pattern using a Publish/Subscribe pattern or one of the Publish/Subscribe variants.



**Figure 3. Message Bus pattern using a Publish/Subscribe pattern or one of the Publish/Subscribe variants**

Table 1 shows the responsibilities and collaborations that are associated with the message bus.

**Table 1: Message Bus Responsibilities and Collaborations**

| **Responsibilities** | **Collaborations** |
| --- | --- |
| –Provides a common set of message formats to the participating applications. –Transports messages from the sender to the other applications that are connected to the bus. | –Senders tag outgoing messages and pass them to the bus. –Receivers inspect the incoming messages and discard the messages that are of no interest to any application. |

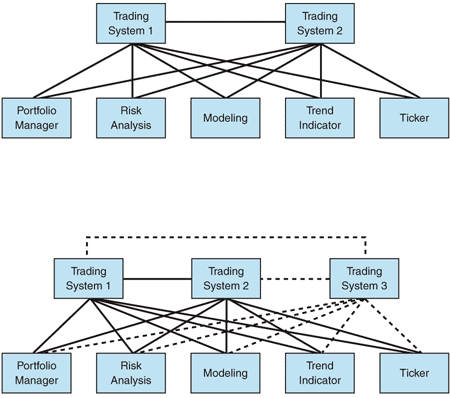
Choosing a message bus for communication between the components of an integration solution lowers the coupling, but it introduces other problems. The following are some questions you should ask when considering a message bus for an integration solution:

* Bus latency. How long does it take the message bus to deliver a message to all the applications that are connected to it? What happens if a sender tries to pass a message to the bus before the bus completes message delivery?
* Bus contention. How do you prevent some applications from monopolizing the message bus? If some applications monopolize the message bus, other applications cannot use it.
* Bus arbitration. How do you deal with multiple applications that want to pass messages to the message bus at the same time?

## Example

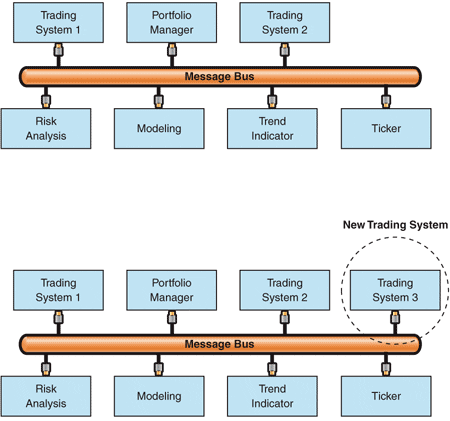
Consider an integration solution that integrates two trading systems, a portfolio manager, a risk analysis application, a modeling application, a trend indicator, and a stock ticker. The trading systems communicate with each other whenever they process a transaction. They also send updates to the other applications.

A point-to-point configuration requires individual connections between each trading system and all six applications. In other words, integrating the participating applications involves 11 connections between the participating applications. Figure 4 shows this topology (top) and the connections that are required to extend the solution to include an additional trading system (bottom). The dotted lines represent the new connections.



**Figure 4. Trading applications that use point-to-point connectivity**

A message bus reduces the number of connections between the trading applications; Figure 5 (top) shows this topology. As you can see from the figure, each trading application has a single connection to the bus. Each trading system is unaware of how many applications are interested in its transactions. With this topology (bottom), adding a new trading system requires a single connection and does not affect the existing applications.



**Figure 5. Trading systems communicating through a message bus**

## Resulting Context

When considering integration through a message bus, you should weigh the following benefits and liabilities that are associated with it:

### Benefits

* **Improved modifiability**. Each application has a single connection to the message bus instead of multiple dedicated connections to each of the other applications. Adding or removing an application has no impact on the existing applications. In addition, upgrading the bus interface does not require changing any applications as long as the messages remain compatible with existing ones. For example, this is the case when you add new message types.
* **Reduced application complexity**. The message bus transports messages between senders and receivers. Senders no longer interact with all the receivers that they need to send messages to.
* Improved performance. There are no intermediaries between the communicating applications. Communication latency depends only on how fast the message bus can move messages.
* Improved scalability. Adding applications has a constant low cost. In addition, you can add applications to the message bus until the message bus is saturated. A message bus is saturated when it cannot keep up with the data that it has to move between applications.

### Liabilities

* **Increased complexity**. Integrating through a message bus increases the complexity of the integration solution for the following reasons:
  + **Architectural mismatch**. The applications of an integration solution typically make conflicting architectural assumptions [Garlan95]. Designing the message bus interface and solving the mismatch around the data model is a difficult endeavor.
  + **Message bus access policies**. Communication through a shared resource such as a message bus requires you to implement policies that ensure fair access and that resolve concurrency conflicts.
* **Lowered modifiability when the bus interface breaks compatibility**. Changing the message bus interface in a way that breaks compatibility typically affects all the applications that use the bus. In other words, the bus interface represents an extreme example of a published interface. Designing it requires foresight.
* **Lowered integrability**. All the applications that are hooked to the message bus must have the same message bus interface. Applications that have incompatible interfaces cannot use the message bus. Because the message bus interface includes a common set of command messages, message schemas, and shared infrastructure, these elements together define a common subset that may somewhat restrict the operation of the participating applications.
* **Lowered security**. A message bus that uses the Broadcast-Based Publish/Subscribe pattern reaches all the applications that are connected to the bus, regardless of the applications that the message is intended for. Broadcasting to all participants may not be acceptable if the messages contain sensitive data.
* **Low tolerance for application unavailability**. The receiver must be able to process messages when the sender passes the messages to the bus. This solution does not tolerate receiver downtime. In addition, it does not provide direct support for disconnected operation.

## Security Considerations

Before you use a message bus that uses Broadcast-Based Publish/Subscribe, you should consider whether this configuration meets your security requirements. The applications that are connected to the bus receive every message that goes through the message bus. Participants that require a private conversation must encrypt their communication. Also, applications that communicate through the message bus do not have intermediate components between them. In other words, no physical component exists for mapping between different security contexts. Consequently, this configuration is appropriate when the security context is managed through impersonation.

## Operational Considerations

When a message bus becomes saturated, message delivery may take a long time or even fail. Saturation could occur after you add new applications or after you make changes to the communication profile of existing applications. For example, changes to the communication profile include changes in the message size and rate. Because both situations are common in bus-centered integration solutions, it is important to prevent saturation. This translates into monitoring the operation of the message bus and proactively keeping the message volume below the maximum capacity of the message bus.

<https://docs.microsoft.com/en-us/previous-versions/msp-n-p/ff647328(v=pandp.10)?redirectedfrom=MSDN>

# Message Broker

## Context

You have an online store that integrates several systems, such as the Web-based front-end system, the credit card verification system, the retail system, and the shipping system. The control flow usually originates in the front end. For example, a customer placing an order causes the online store to send a request message to the credit card verification system. If the credit card information is validated, the online store sends request messages to the various retail systems, depending on the ordered items. An order for books translates into a purchase order message for the book retailer; an order for electronics translates into a purchase order message for the electronics retailer; and an order for gardening supplies translates into a purchase order message for the home and garden supplier.

The control flow could also originate in a retail or shipping system. For example, when a retailer updates a catalog, the retail system sends catalog update messages to the store so that the store can display the new items. When a shipper changes the shipping rates, the shipping system sends a rate update message to the store so that the store can compute the correct shipping charges. Similarly, when a shipper changes pickup times, the shipping system sends update messages to all the retailers the system serves so that they can have the shipments ready in time.

## Problem

How do you integrate applications without enforcing a common interface and also allow each application to initiate interactions with several other applications?

## Forces

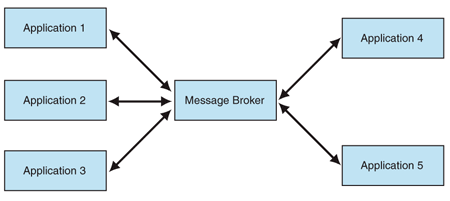
To integrate applications without changing their interfaces, you must balance the following forces:

* Point-to-point integration requires a large number of connections between applications. Many connections usually translate into many interfaces.
* [Message Bus](https://docs.microsoft.com/en-us/previous-versions/msp-n-p/ff647328%28v%3dpandp.10%29) integration facilitates adding new applications, but it requires a common bus interface. Because integration solutions usually involve applications that have proprietary interfaces provided by multiple vendors, Message Bus integration is difficult.
* Run-time control of qualities such as availability and performance may require dynamic reconfiguration.
* The applications in an integration solution could have conflicting quality of service (QoS) requirements.
* The applications in an integration solution could belong to different security realms.

## Solution

Extend the integration solution by using Message Broker. A message broker is a physical component that handles the communication between applications. Instead of communicating with each other, applications communicate only with the message broker. An application sends a message to the message broker, providing the logical name of the receivers. The message broker looks up applications registered under the logical name and then passes the message to them.

Communication between applications involves only the sender, the message broker, and the designated receivers. The message broker does not send the message to any other applications. From a control-flow perspective, the configuration is symmetric because the message broker does not restrict the applications that can initiate calls. Figure 1 illustrates this configuration.



**Figure 1. A message broker mediating the collaboration between participating applications**

The message broker can expose different interfaces to the collaborating applications, and it can translate messages between these interfaces. In other words, the message broker does not enforce a common interface on the applications.

Prior to using a message broker, you must register the applications that receive communications so that the message broker can dispatch requests to them. The message broker may provide its own registration mechanism, or it may rely on an external service such as a directory.

Placing the message broker between the sender and the receiver provides flexibility in several ways. First, the message broker allows the integration solution to dynamically change its configuration. For example, if an application must be shut down for maintenance, the message broker could start routing requests to a failover application. Likewise, if the receiver cannot keep up with the incoming messages, the message broker could start load balancing between several receivers.

Second, the message broker can choose between applications that have different QoS levels. This resembles the dynamic configuration, but the message broker selects the application based on specified criteria. For example, an application for premium accounts may fulfill requests quickly, but an application for general use may have a longer processing time.

Third, the message broker allows the sender and the receiver to reside in different security realms. In other words, the message broker can reside on the boundary between two security realms and bridge requests between those two realms. Table 1 shows the responsibilities and collaborations of a message broker.

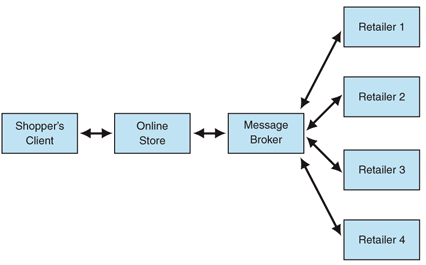
**Table 1: Message Broker Responsibilities and Collaborations**

| **Responsibilities** | **Collaborations** |
| --- | --- |
| –Receive message | –Senders: applications that send messages to the message broker. |
| –Determine the message recipients and perform the routing | –Receivers: applications that receive messages from the message broker |
| –Handle any interface-level differences |  |
| –Send the message to the recipients |  |

## Example

Consider an online store that allows shoppers to browse a variety of retail catalogs and to place orders. When an order is placed, the online store groups the shopping cart items by retailer and places appropriate orders with each retailer. As each retailer fulfills and ships the order, it sends the online store a tracking number. The online store updates its records so that this information is presented on the Check Order Status page. If the customer has configured e-mail alerts, the store also sends an e-mail message that contains the tracking information.

The online store that is illustrated in Figure 2 uses a message broker to communicate with the individual retailers. The broker knows how to reach each retailer and how to place orders, to cancel orders, and to check the order status. Likewise, the retailers communicate with the broker when they send tracking numbers. Each retailer must know how to reach the broker and how to send the tracking number. In other words, both the store side and the retailer side can initiate a communication, and the data flows in both directions.



**Figure 2. Online store communicating with retailers through a message broker**

## Resulting Context

The decision to use a message broker for integration entails balancing the benefits of removing inter-application coupling against the effort associated with using the message broker. Use the following benefits and liabilities to evaluate the balance:

### Benefits

* **Reduced coupling**. The message broker decouples the senders and the receivers. Senders communicate only with the message broker, and the potential grouping of many receivers under a logical name is transparent to them.
* **Improved integrability**. The applications that communicate with the message broker do not need to have the same interface. Unlike integration through a bus, the message broker can handle interface-level differences. In addition, the message broker can also act as a bridge between applications that are from different security realms and that have different QoS levels.
* **Improved modifiability**. The message broker shields the components of the integration solution from changes in individual applications. It also enables the integration solution to change its configuration dynamically.
* **Improved security.** Communication between applications involves only the sender, the broker, and the receivers. Other applications do not receive the messages that these three exchange. Unlike bus-based integration, applications communicate directly in a manner that protects the information without the use of encryption.
* **Improved testability**. The message broker provides a single point for mocking. Mocking facilitates the testing of individual applications as well as of the interaction between them.

### Liabilities

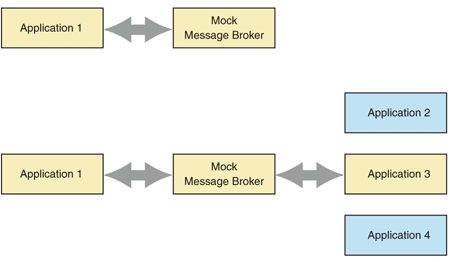
* **Increased complexity**. Communicating through a message broker is more complex than direct communication for the following reasons:
  + The message broker must communicate with all the parties involved. This could mean providing many interfaces and supporting many protocols.
  + The message broker is likely to be multithreaded, which makes it hard to trace problems.
* **Increased maintenance effort**. Broker-based integration requires that the integration solution register the applications with the broker. Bus-based integration does not have this requirement.
* **Reduced availability**. A single component that mediates communication between applications is a single point of failure. A secondary message broker could solve this problem. However, a secondary message broker adds the issues that are associated with synchronizing the states between the primary message broker and the secondary message broker.
* **Reduced performance**. The message broker adds an intermediate hop and incurs overhead. This overhead may eliminate a message broker as a feasible option for solutions where fast message exchange is critical.

## Testing Considerations

A mock message broker can receive requests and send back canned responses. In effect, the mock message broker allows system testers to verify individual applications without removing the individual applications from the integration solution.

Likewise, a mock message broker that emulates some of the message broker's functionality allows testers to verify and to profile the interplay between a few applications, for example, a subset of the integration solution. In other words, a mock message broker allows you to test individual applications and to test subsystems.

Figure 3 shows these configurations; the shading indicates the areas under test.



**Figure 3. Using a mock message broker to test applications and subsystems**

## Security Considerations

Integration by using a message broker places a component between senders and receivers. On one hand, this configuration accommodates management of the security context through consolidation as well as impersonation. On the other hand, the message broker represents a central point of attack. Compromising the message broker compromises the communication between all the applications that use it.

## Operational Considerations

The message broker can dynamically change the configuration. It can direct messages to a failover application if necessary or perform load balancing between applications, or the message broker can do both.

## Known Uses

Broker-based integration products such as Microsoft BizTalk Server 2004 extend the traditional broker functionality with additional features. For example, BizTalk Server 2004 provides orchestration services, messaging services, and other features associated with these services such as business activity monitoring, transaction and exception handling, correlation, and graphic editing.

## Variants

A message broker variant trades ease of integration for performance. The performance-optimized message broker looks up the receiver and then connects it to the sender, thus allowing the sender and the receiver to communicate directly. The direct connection eliminates the performance penalty that is associated with an intermediary between the communicating parties. However, this performance optimization only works if the sender and the receiver have the same interface.

<https://docs.microsoft.com/en-us/previous-versions/msp-n-p/ff648849(v=pandp.10)?redirectedfrom=MSDN>