一种自协调的分布式DDS系统

摘要 在传统系统设计中，一般设计人员只考虑以应用为中心，针对系统所要完成的工作进行程序的编写，数据是为应用而服务。这样做的缺点就是应用与应用之间因为数据相关联导致强耦合，例如在一个进程间通信的系统中，如果以每一个进程为中心，则需要在每一个进程中编写一个进程间通信的接口的实现，这样不仅效率低，而且会使系统间进程的通信关系错综复杂，耦合性增强，这样不利于系统的维护和升级。现代应用系统的设计一般以分层，模块化为主，模块化的基础是信息间的高效实时通信保障，因此设计上应该围绕着数据来进行，即以数据为中心(Data-Centric)。本文提出了一种节点间自协调机制的数据分发服务(Data Distribution Service, DDS)系统，旨在为分布式的进程间提供一种实时通信的中间件(Middleware)，屏蔽所有的通信细节。

关键词: 进程间通信(IPC) 数据为中心 数据分发服务

1. 介绍
2. OMG标准
3. 实现方法
4. 底层通信原理

进程间通信(Inter-Processes Communication)原理主要有以下几种。

1. 信号量

在操作系统中存在共享资源，临界资源，临界区。其中共享资源指的是在多道程序系统中，多个进程都需要访问到的一些资源。临界资源指的是在这些共享资源中一次只能有一个进程可以访问的资源，比如打印机，一次只能有一个进程来使用它。临界区是指各个进程访问这些临界资源的代码。因此我们必须保证临界区在同一时间只有一个进程在访问它。而信号量就是用来协调进程对共享资源访问的。

信号量是一个特殊的变量，程序对其访问都是原子操作，且只允许对它进行等待和发送信息操作。

1. 共享内存

共享内存也是一段内存，它和其它普通内存的区别在于其它内存只属于单个进程，而共享内存却可以被多个进程访问。不同进程之间的共享内存通常安排为同一段物理内存，进程可以将同一段共享内存连接到他们自己的地址空间中，所有进程都可以访问共享内存中的地址，如果某个进程向共享内存写入数据，所做的改动将立刻影响到可以访问同意共享内存的任何其他进程。

1. 消息队列

提供了一种从一个进程向另一个进程发送一个数据块的方法。消息队列是消息的链接表，存放在内核中并由消息队列标识符标识。 每个数据块都被认为含有一个类型，接收进程可以独立地接收含有不同类型的数据结构。

消息队列的相关操作有创建消息队列，利用消息队列发送消息，接收消息队列中的数据，以及删除消息队列。

1. Socket

Socket通信是目前最流行的进程间通信方式，主要在于它可以实现跨平台，跨设备的进程间交互方式。一般来说Socket需要借助于TCP/IP协议栈来实现进程间的交互通信。

1. 通信架构

本项目使用socket作为底层通信方式，在UDP协议的基础上实现一个进程间消息通信的中间件。根据DDS标准对每一种消息定义一个topic，发送方称之为publisher，接收方称之为subscriber。通信各方通过topic进行信息关联。每一个通信的进程称之为Node。简单的发布订阅时通信结构如下：

Topic

Topic

Topic

图1.通信结构图

对于进程中的发布者来说，它只管发布消息，至于订阅方是否收到消息或者订阅方是否存在均不关心，订阅者与发布者之间仅通过Topic来进行关联，即订阅方只关系该Topic下的所有信息，其它信息均不关心，至于Publisher是否存在也不关心，这样就做到了信息的发送方和订阅方进程间解耦合，发布与订阅没有直接联系，进程间相互不会受到干扰。

1. 节点间发现协议

传统的RTSP(Real-Time Subscriber Publisher)通信框架例如ROS（Robot Operating System），都是需要一个主节点(Master Node)来对所有的节点进行信息注册和管理，每新进一个节点，节点都会向主节点发送一条消息，说明自己是何种类型的节点(Subscriber Node or Publisher Node),自己的IP和端口号，主题是什么。主节点收到这些信息之后会将这些信息保存下来，并与已存在的节点对其主题进行匹配，如果一个发布节点和一个订阅节点的主题相匹配上，则将订阅节点的IP和端口号告诉发布节点，这样发布节点直接通过IP和端口与订阅节点建立TCP连接，从而进行信息间交互。这里的主节点主要起到一个协调的作用，类似于RPC(Remote Procedure Call)中的注册中心(Register Center)。其通信框图如下所示：

1.发布节点向主节点注册自身信息

2.订阅节点向主节点注册自身信息

5.订阅节点向主节点注册自身信息

3.主节点将相同主题的发布节点端口和IP地址告知订阅节点

4.订阅节点根据端口和地址直接与发布节点建立连接

6.主节点将相同主题的发布节点端口和IP地址告知订阅节点

7.订阅节点根据端口和地址直接与发布节点建立连接

图2. ROS节点间通信过程

使用主节点作为节点间协调的方法有以下几个缺点：

1. 节点之间的通信强烈依赖于主节点的协调工作，意味着一旦主节点因为意外而挂掉，整个通信结构都将无法通信，对于车载终端来说，这是无法容忍的。
2. 单独的运行的主节点同样会消耗cpu和网络资源，在一些嵌入式系统中，这种消耗是不必要的。

为了避免这些问题，我们提出了一种节点间自协调的DDS框架。为了删除主节点，我们设置了一种节点间发现协议，帮助发布节点和订阅节点之间能够根据主题相互间发现对方，从而建立起通信连接。节点间发现协议的核心思想是在系统中的每一个节点中维护一张节点间信息表(Nodes Information Table),每当有一个节点加入系统或者从系统中删除时，节点中的所有节点都会第一时间更新此表，每个节点会根据这个表来找到对应的具有相同主题的节点并自己与之建立通信连接而不再依赖于主节点。通过此协议可以实现此通信框架的去中心化(Decentralized),克服中心节点所带来的缺陷。

接下来将详细介绍节点间发现协议的工作原理。

* 1. 每一个节点在启动时会作为一个UDP Client向整个网络中的固定端口广播一条topicInfo信息，topicInfo消息帧的格式如下；

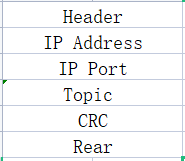


图3. TopicInfo消息帧格式

假设第一个节点为一个发布信息节点加入到系统中时，如果该节点在一段时间内没有收到任何针对此消息的回复则可以判断出自己是整个系统中的第一个节点，此时将自己设置为UDP Server监听端口是否有新节点的加入同时更新自己的节点间信息表，将自己的IP地址，端口号以及主题信息加入到表中，并记录自己的节点序号(Node Number)为1，代表自己是系统中的第一个节点。示意图如下：

信息表

IP address

IP port

Topic1

图4.节点1加入后系统状态

* 1. 当第二个节点加入时(假设它是一个订阅节点)，同样的作为一个UDP Client向固定端口发送一条topicInfo，此时系统中已有的一个节点是作为UDP Server存在的，它能接收到此信息并回复表示自己已收到topic信息。

信息表

Node1 IP address

Node1 IP port

Node1 Topic1

信息表

NULL

topicInfo

response

图5.节点2的发现过程

当第二个节点在收到第一个节点的Response之后，则二者之间开始更新各自携带的节点信息表。双方更新完节点间信息表之后节点2发现自身的Topic与节点1相同，并且各自的节点类型相匹配，于是可以建立起连接通道，此时节点2可以接收节点1的传输的消息。

信息表

Node1 IP address

Node1 IP port

Node1 Topic1

Node2 Ip address

Node2 Ip port

Node2 Topic1

信息表

Node1 IP address

Node1 IP port

Node1 Topic1

Node2 Ip address

Node2 Ip port

Node2 Topic1

messages

图6. 节点间建立连接

* 1. 当第三个节点加入时（假设它是一个订阅节点），依然作为一个UDP Client广播一个topicInfo消息帧。同样的节点1作为UDP Server接收到消息后会响应节点3的消息帧，表示自己已经接收到节点3的节点信息，此时会对系统中所有节点进行节点间信息表的更新。当节点间信息表更新完毕后，节点3的Topic和节点1的Topic相匹配上，同样建立起连接。

信息表

Node1 IP address

Node1 IP port

Node1 Topic1

Node2 Ip address

Node2 Ip port

Node2 Topic1

Node3 Ip address

Node3 Ip port

Node3 Topic1

messages

信息表

Node1 IP address

Node1 IP port

Node1 Topic1

Node2 Ip address

Node2 Ip port

Node2 Topic1

Node3 Ip address

Node3 Ip port

Node3 Topic1

信息表

Node1 IP address

Node1 IP port

Node1 Topic1

Node2 Ip address

Node2 Ip port

Node2 Topic1

Node3 Ip address

Node3 Ip port

Node3 Topic1

messages

图7. 节点3的加入

* 1. 为了保障节点之间能够发现对端节点的删除。需要维持节点间通信的长连接(Long Connection)。当系统中某个节点因为主动关闭或者意外掉线时需要及时知晓并及时更新系统中各个节点的连接信息表，删除已经关闭或者掉线的节点信息。维持长连接的一个措施是采用心跳机制(Heartbeat Mechanism)。所谓“心跳”就是定时发送一个自定义的结构体（心跳包或心跳帧），让对方知道自己“在线”。 以确保链接的有效性。一旦发送方发送的心跳包对端没有响应或者接收方一段时间内没有心跳包，则可以判断出节点已经掉线，因此采用心跳机制的系统间通信结构如下所示：

图8.心跳机制

信息表

Node1 IP address

Node1 IP port

Node1 Topic1

Node2 Ip address

Node2 Ip port

Node2 Topic1

Node3 Ip address

Node3 Ip port

Node3 Topic1

messages

信息表

Node1 IP address

Node1 IP port

Node1 Topic1

Node2 Ip address

Node2 Ip port

Node2 Topic1

Node3 Ip address

Node3 Ip port

Node3 Topic1

信息表

Node1 IP address

Node1 IP port

Node1 Topic1

Node2 Ip address

Node2 Ip port

Node2 Topic1

Node3 Ip address

Node3 Ip port

Node3 Topic1

messages

Heartbeats

Heartbeat

* 1. 当节点2掉线时，节点1可通过心跳机制及时检测到节点2已经退出，此时整个系统中所有的节点都要重新更新自己的节点间信息表，删除关于节点2的信息，并且节点1与节点2之间的连接将断开。此时系统间节点结构如下所示：

信息表

Node1 IP address

Node1 IP port

Node1 Topic1

Node3 Ip address

Node3 Ip port

Node3 Topic1

信息表

Node1 IP address

Node1 IP port

Node1 Topic1

Node3 Ip address

Node3 Ip port

Node3 Topic1

messages

Heartbeat

图9. 删除节点2信息

* 1. 当节点1掉线时，这时节点3同样的可以通过心跳机制检测到节点1已经掉线，此时节点3除了要将自身的节点间信息表更新外，还需要将自己从UDP Client重置为UDP Server，以代替节点1来检测系统中是否有其它节点的接入或者删除。

信息表

Node3 Ip address

Node3 Ip port

Node3 Topic1

图10. 删除节点1信息

1. 节点间信息通信

在通过节点间发现协议找到每个节点找到与之主题相对应的节点之后，接下来就可以进行消息的传输，具体传输方法可以使用共享内存，socket，消息队列等等方式。具体通信方式可选，一般在本机之内采用共享内存或者消息队列，在主机之间采用socket。通信的具体过程如下：

* 1. 消息格式定义

传输消息的格式可随时自定义，定义方式采取结构体方式。例如IMU的消息格式可定义为：

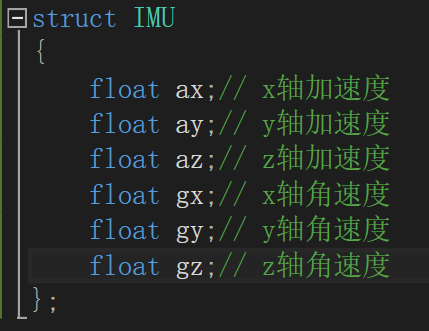


图11. IMU消息格式

* 1. 消息序列化(Serialize)

消息序列化是指将消息转化为二进制流，方便在网络中进行传输。常用的序列化方式有xml，jason编码方式以及谷歌的变长编码库protobuf等等。在此库中采用直接编码方式来实现消息的序列化。例如一个int型数据在计算机中以二进制补码格式编码，占据4个字节。消息传输时直接将这4个字节数据发送给对端，对端在接收到这4个字节数据之后通过反序列化(Deserialize)方式还原成原来的int型数据。

* 1. 消息传输

当对要发送的数据进行序列化之后，即可通过共享内存或者socket通信的方式来进行消息传输。我们将操作系统对共享内存或者socket提供的API系统调用函数进行进一步的封装。应用程序不直接调用系统接口，提高效率。

* 1. 使用IO复用机制

在通信过程中，我们不使用while循环阻塞接收消息，而是使用IO复用模型。IO多路复用是一种同步IO模型，可以实现通过在操作系统层面监听文件句柄，一旦某个文件句柄就绪，即一个事件(Event)就绪，就能够通知应用程序进行相应的读写操作。

linux下最常用的IO复用模型有select，poll和epoll，本框架采取epoll模型。

* 1. 消息回调机制

我们在IO复用机制的基础上实现了一个消息回调(Callback)处理机制。对于每个订阅者，在收到发布者发布的消息之后，将立即调用注册的回调函数，实现对消息的实时并且及时处理。