一种分布式的进程间通信系统的设计

一、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；

二、前言

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。

数据分发服务（DDS）是一个由对象管理组（OMG）发布的以数据为中心的中间件协议和 API 标准。DDS 集成系统中的各个组件，提供低延迟数据连接、高可靠性以及高可扩展体系结构。在分布式系统中，中间件是位于操作系统和应用程序之间的软件层。它使系统的各个组件能够更轻松地通信和共享数据。它简化了分布式系统的开发，让软件开发人员专注于其应用程序的特定用途，而不是在应用程序和系统之间传递信息的机制。

分布式系统指通过网络互连，可协作执行某个任务的独立计算机集合。在实际生产生活中，由于现代计算机系统程序通常运行在操作系统之上，大型任务通常需要分解成许多个子任务进行，每一个子任务可以分别对应一个计算机进程，进程与进程之间通过某种通信方式实现信息的交互，从而使任务能够更有效地执行并且能大幅度提高开发者的工作效率。由于近些年来网络通信技术的飞速发展，计算机网络已经相当普及，因此基于网络通信的进程间通信方式越来越受到欢迎，这种分布式的进程间通信方式，克服了地理位置之间的障碍，克服了距离之间的阻隔，使得多个计算机之间协同解决大型问题成为可能。

为了实现分布式的计算机进程间的通信工作，本文设计了一种基于 DDS 协议的计算机进程间通信系统。本系统采用了 DDS 标准，实现了以数据为中心的发布/订阅(Data-CentricPublish-Subscribe)机制，DDS 将分布式网络中传输的数据定义为主题(Topic),将数据的产生对象定义为发布者(Publisher),数据的接收对象定义为订阅者(Subscriber),两者执行的动作即发布(Publish)和订阅(Subscribe)，并将每一个进程实体称之为节点(Node)。使用订阅/发布机制可有效实现分布式的多进程通信中的数据解耦合,即节点间数据的发布和订阅是分开的，互不关系对方是否存在，只需通过主题(topic)来进行关联即可。此外，本系统在 UDP 协议基础上实现了一个节点间发现协议,可实现分布式网络中的节点间的自组网工作，包括网

络节点的增添和删除管理，可实现一对一，一对多和多对多的进程间通信方式，并且只需要使用简单的几个接口函数即可实现。

三、相关工作

四、设计方法

1. 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。

2) 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。

3) 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。

4) 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。

2. 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：

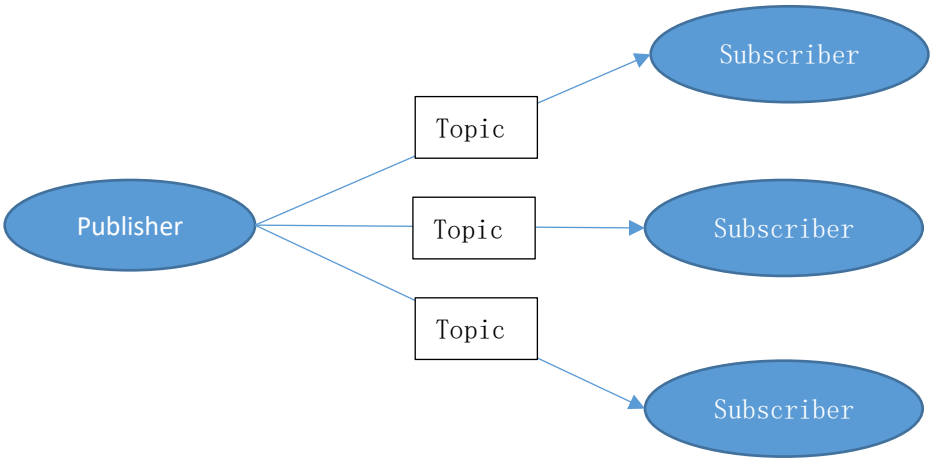

图 1.通信结构图

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。

3. 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：

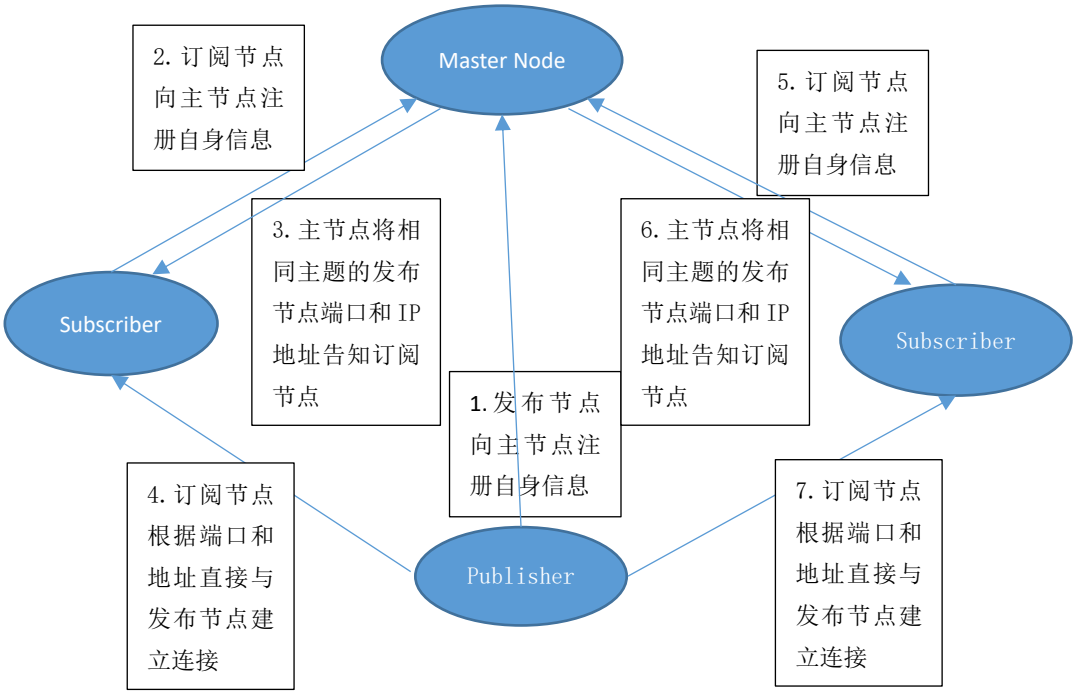

图 2. ROS 节点间通信过程

使用主节点作为节点间协调的方法有以下几个缺点：

a. 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。

b. 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。

3.1 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；

Header
IP Address
IP Port
Topic
CRC
Rear

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：

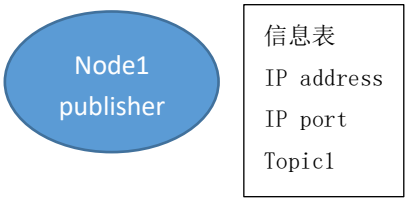

图 4. 节点 1 加入后系统状态

3.2 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。

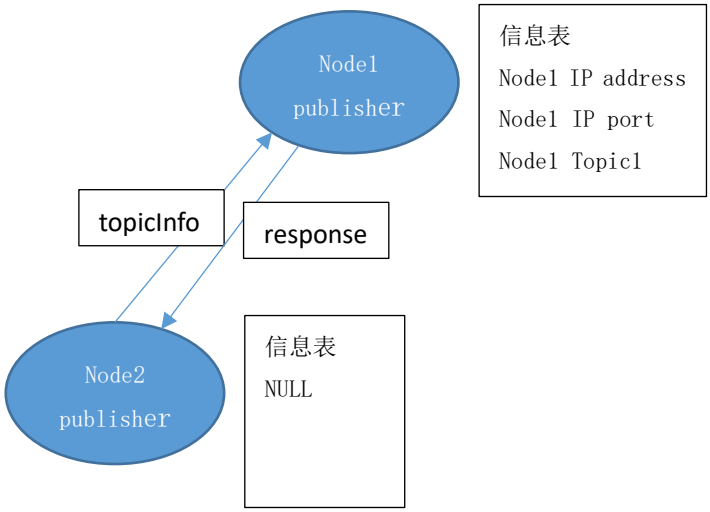

图 5. 节点 2 的发现过程

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。

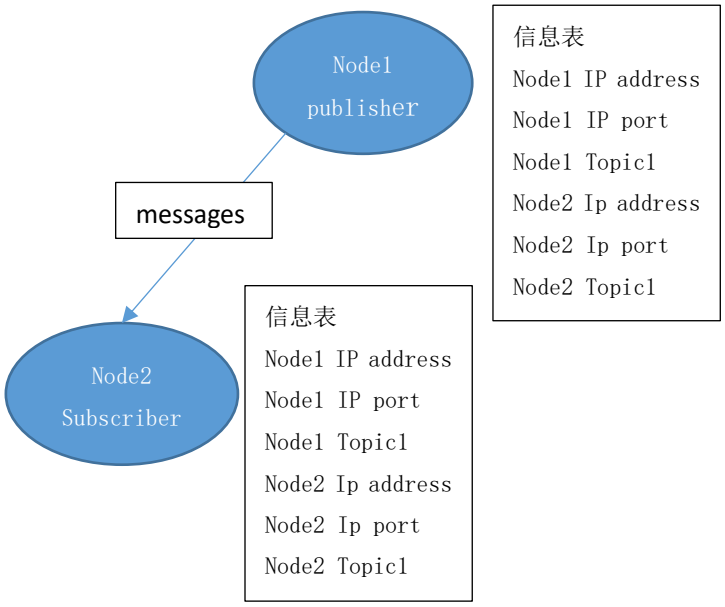

图 6. 节点间建立连接

3.3 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。

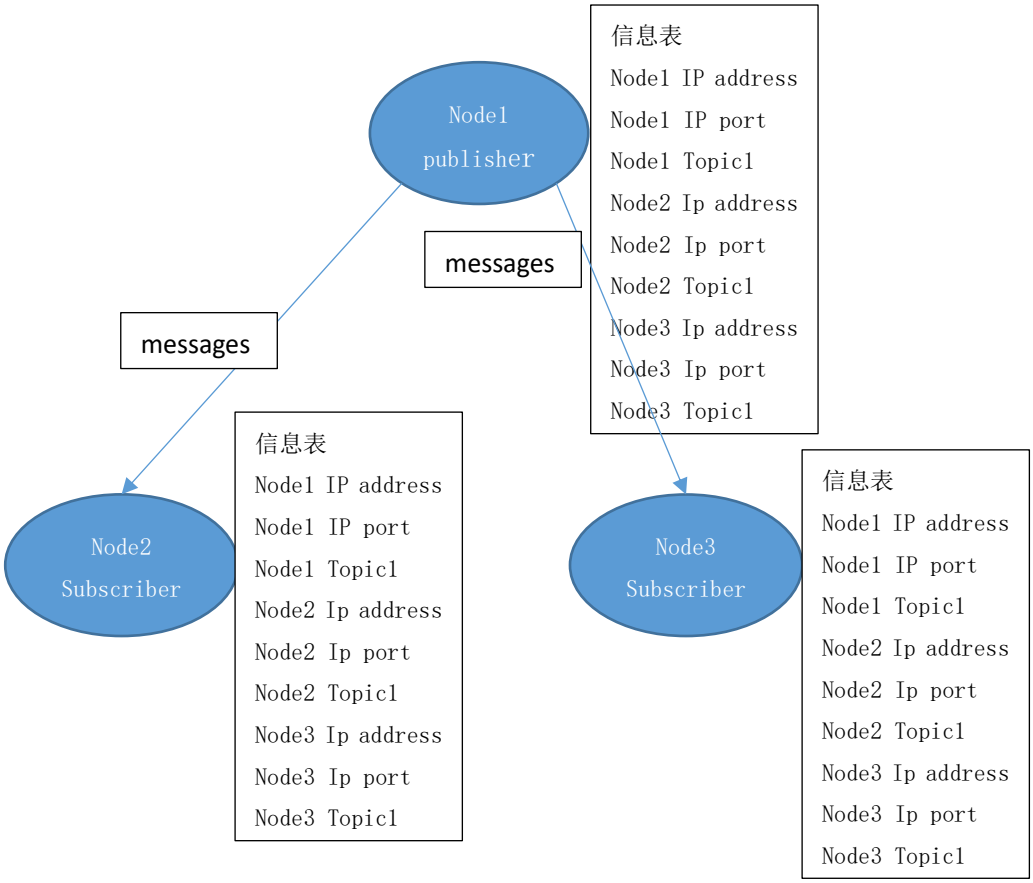

图 7. 节点 3 的加入

3.4 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：

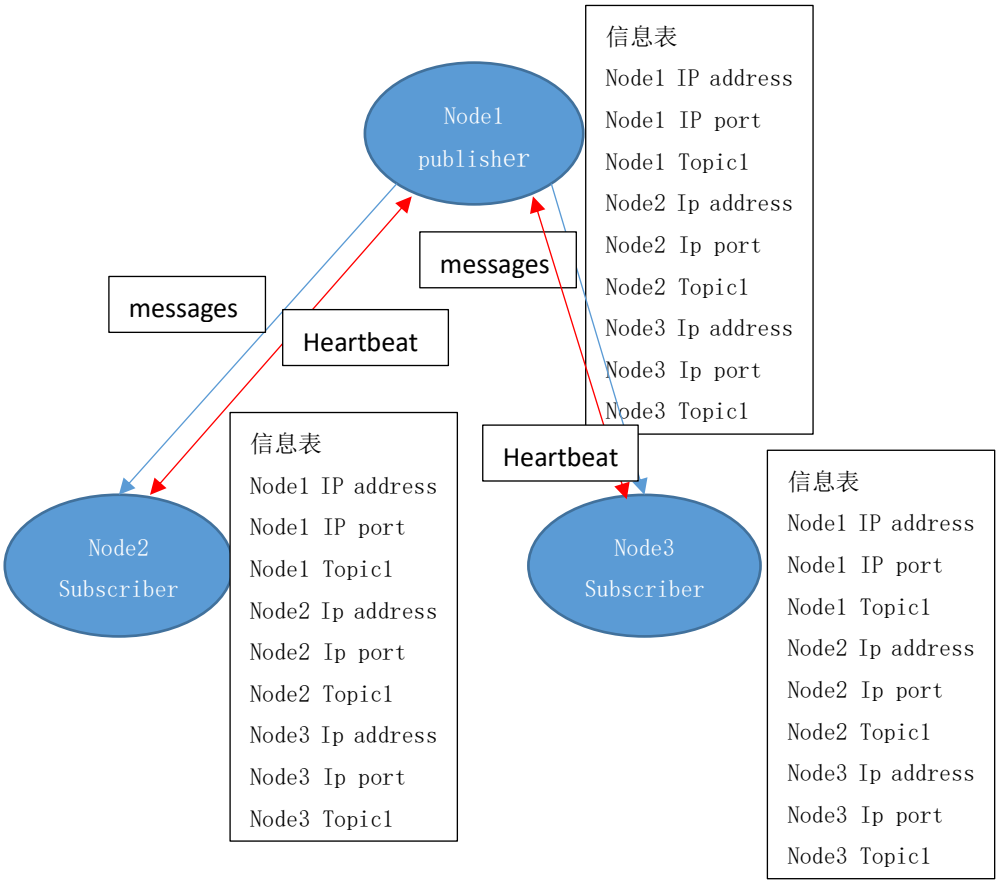

图 8. 心跳机制

3.5 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：

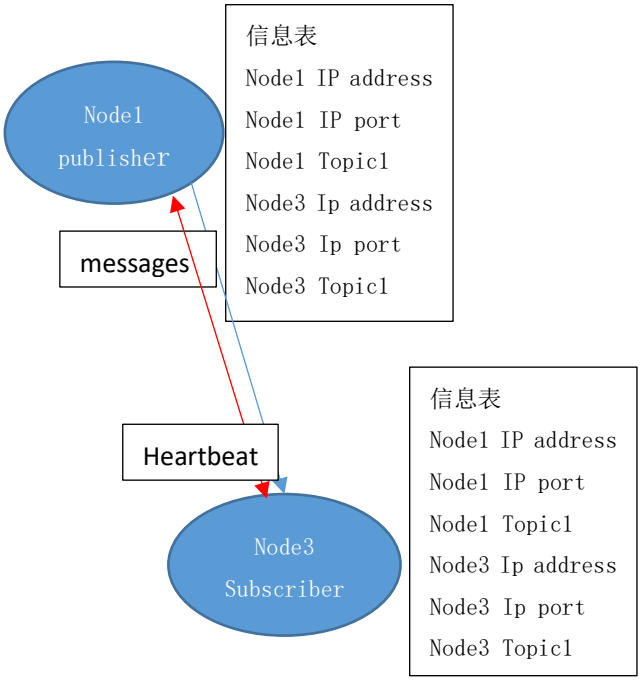

图 9. 删除节点 2 信息

3.6 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。

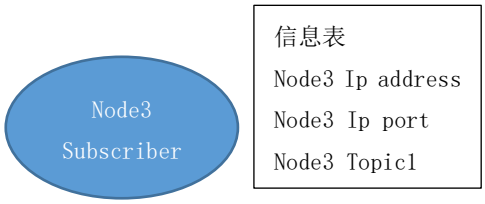

图 10. 删除节点 1 信息

4. 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：

4.1 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。

4.2 IO 复用技术

IO 复用技术是一种同步 IO 模型，可以用来实现一个线程监听多个文件句柄，一旦某个文件句柄就绪，就能够通知应用程序进行相应的读写操作，没有文件句柄时候程序线程就会阻塞，让出 CPU 的所属权，本系统所采用的 SOCKET 通信方式同样属于计算机 IO 的一部分，因此同样可以使用 IO 复用技术来提高系统的执行效率，尤其是对于发布者来说，通常一个发布者往往要对应很多个订阅者，维持长连接以及消息的交互中需要使用 IO 复用技术来提高通信的时间效率。在 linux 环境下主要有 select、poll 和 epoll 三种 IO 复用技术模型。本系统主要采用 epoll 模型。

4.3 Epoll 反应堆模型

Epoll 的底层模型是使用红黑树实现的，在 linux 操作系统下提供了三个接口：epoll_create(),epoll_ctl()以及 epoll_wait()。执行 epoll_create()时，操作系统初始化一个红黑树和一个就绪链表，红黑树的每一个节点即要挂载的监听事件。epoll_ctl()用于将事件注册到红黑树上并向内核注册回调函数，用于当中断事件来临时向准备就绪链表中插入数据。epoll_wait()用于返回准备就绪链表中的数据。Epoll 的反应堆模型即当事件触发时对事件进行处理并且当事件处理完成之后就立刻将事件从红黑树中摘下，然后添加下一个将要被触发的事件，循环往复，可以提高通信效率。

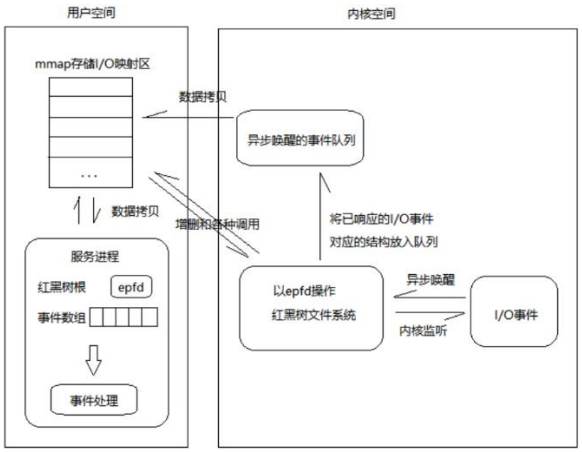

图 epoll 模型

5. 消息间时间戳同步

在分布式的不同的计算机进程之间，消息相互通信的时间戳对齐是一件非常重要的事情，时钟同步是分布式系统的核心技术之一，其目的是维护一个全局一致的物理或逻辑时钟，以使系统中的消息、事件及各节点与时间有关的行为有一个全局一致的解释，以确保节点发送和接收消息在时间逻辑上是完全正确的。在分布式系统中，不同的节点需要精度统一的时钟，由于没有全局的系统节拍，就很难获得精确的时钟同步。而且由于各个进程节点之间发送消息的频率不同，往往没有统一的时钟而使消息不能完全匹配上。

在某些分布式系统种往往采用统一时钟源的方法提供一个全局时钟来做时钟同步，例如

PTP（Precision Time Protocol)协议是 IEEE-1588 中定义的一种精密时钟同步协议，PTP 协议主要针对于相对本地化、网络化的系统，子网较好，内部组件相对稳定的环境设计的。PTP 协议时钟同步机制的系统包含一个主时钟和多个从时钟,时间同步主要通过分别在发送方和接收方对包含时间的信息打时戳,并在接收方根据时戳计算出主从时钟的时间偏差和时间信息在网络中传输的延时来实现。

而在本系统中抛弃了上述这种传统的全局时钟设计方法， 设计了两种消息同步方法：最近时间戳同步方法与时间戳插值方法。

五、系统特点

本系统针对这个问题采用了一个自定义的节点间发现协议(Nodes Discovery Protocol)来消除掉 Master 节点，即在此 PCS 网络中每一个节点都是平等的，任何节点的接入或者删除都不会影响到其他的节点的正常工作，通过 topic 进行匹配和建立通信链接的过程都是节点自己完成的，不需要再像 ROS1 中需要借助于 Master 节点。此方法的实现要点是在每一个节点中都维护一张主题信息表(Topic Information Table)，每当有节点的删除或接入就在整个 PCS 网络中对每一个节点的表都进行更新，确保所有节点所掌握的网络信息都是一样的。

b. 高实时性

本系统未使用任何第三方库，避免了封装带来的效率问题，通过直接调用 linux 操作系统的系统接口来实现所有的操作。进程间的通信主要是通过面向连接的 TCP 协议和共享内存来实现。在主机与主机之间的不同进程间使用 TCP 协议来进行通信，而在主机内部的进程间通信则采用共享内存来实现,这样通信的时延大小也主要取决消息在于信道间传输所花费的时间的大小，而 tcp 协议中解包或者共享内存的一些内存读写操作所消耗的时间影响极小，可满足绝大多数的任务需求。

c. 实现了两种消息时间戳同步的方法

在分布式应用或者多进程间通信过程中，消息同步是一件很重要的事情，本系统采用了两种消息同步的策略。分别是最近时间戳(Approximate TimeStamp Synchronization)同步方法和插值同步(Interpolation Synchronization)方法。其原理可在后续详解。

d. 可自动生成自定义消息格式的头文件

仿造 ROS 的自定义消息，在 PCS 系统中也实现了一个自动生成自定义消息.h 文件的工具，通过在.message 文件中以类 C 语言结构体风格定义一个自定义消息，可通过工具 auto_message_generate 生成一个.h 文件，要想使用自定义消息时，包含这个头文件即可，非常方便。

e. 使用方法简单

由于采用了 pubish/subscribe 的方式，数据之间实现了完全的解耦合，节点与节点之间相互独立，因此在进程间进行通信的时候，每个节点都无需关心对方是否存在，是否接收到数据等等问题，不需要额外编写复杂的应答机制，无需关心底层通信的实现方法。此外要完成基本的通信功能，只需要调用 publish、subscribe 两个函数即可，极其方便快捷。

f. 订阅的回调机制

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。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。