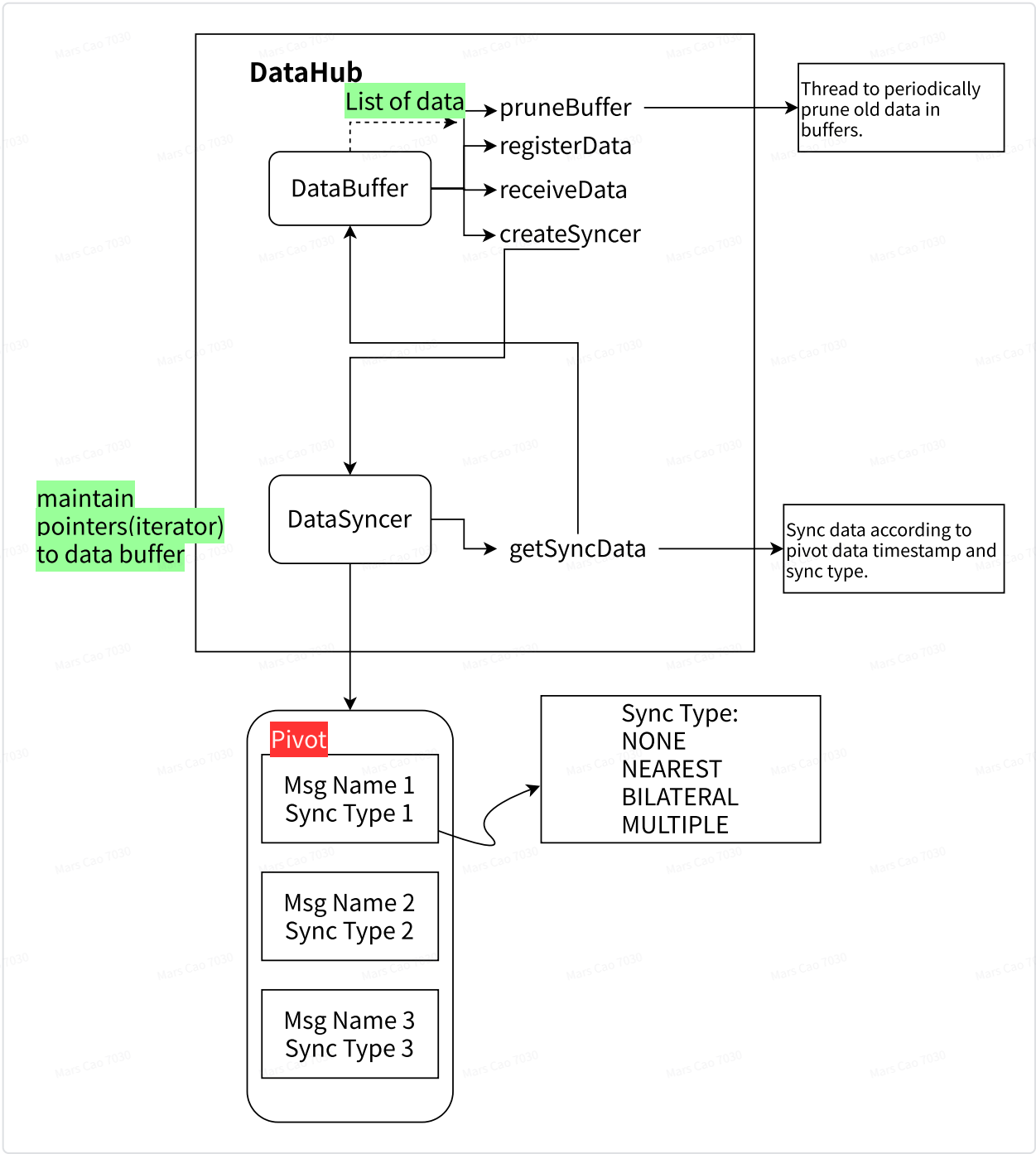
Datahub

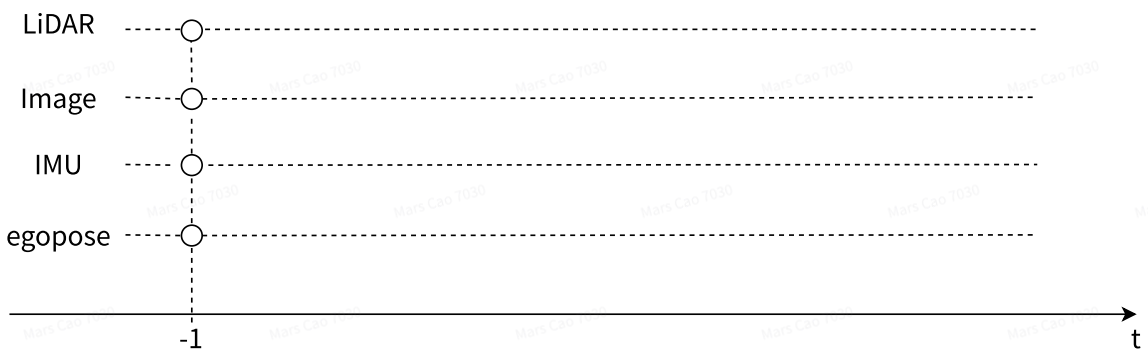
整体框图



算法流程

注册消息类型 Register Message

注册每个消息的list的buff size。同时会为每个链表添加时间戳为-1的头节点。

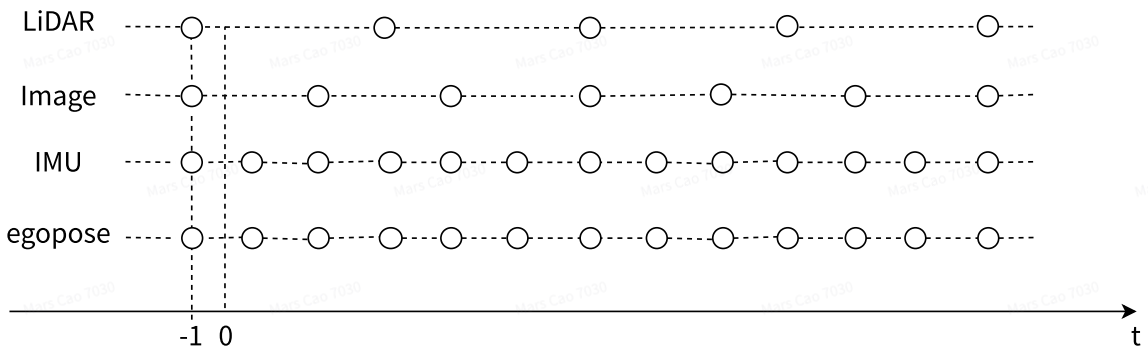


C++

```
DataBuffer buffer_
buffer_.registerMessage("/sensor/IMU", 400);
buffer_.registerMessage("/sensor/Image", 40);
buffer_.registerMessage("/sensor/LiDAR", 20);
buffer_.registerMessage("/egopose", 400);
```

接收消息 Receive Message

调用DataBuffer的receiveMessage方法来给消息链表中添加信息



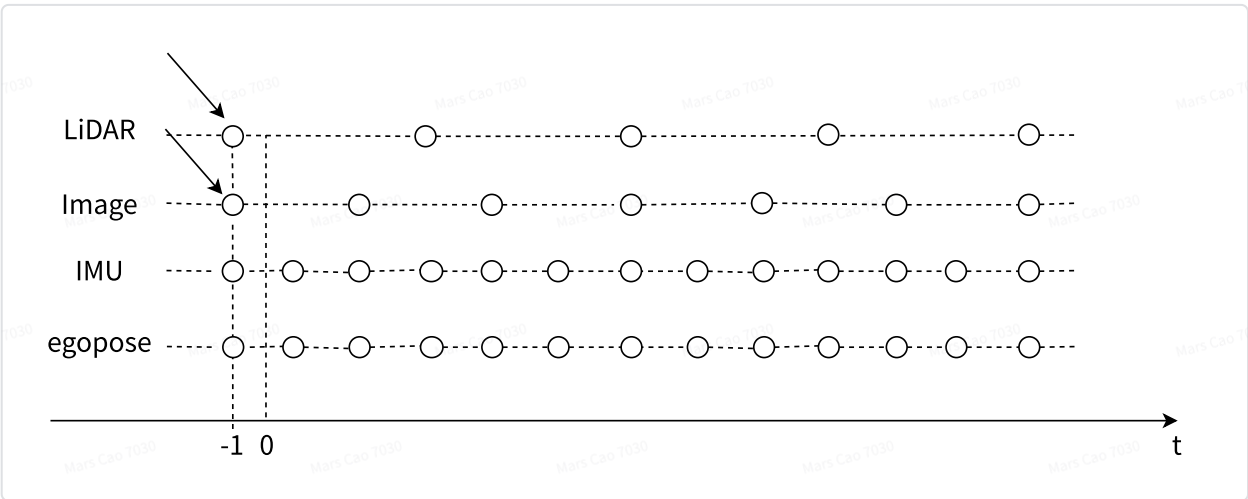
C++

```cpp
//thread 1
Message::Ptr msg(new Message);
msg->name = "IMU";
msg->timestamp =
  std::chrono::time_point_cast<std::chrono::microseconds>(t_n)
      .time_since_epoch()
      .count();
buffer_.receiveMessage(msg);

//thread 2
Message::Ptr msg(new Message);
msg->name = "LiDAR";
msg->timestamp =
  std::chrono::time_point_cast<std::chrono::microseconds>(t_n)
      .time_since_epoch()
      .count();
buffer_.receiveMessage(msg);
```

创建消息同步器 Create DataSyner

通过设计同步方式与同步话题名，来创建对应的同步器。同步器的消息指针一开始指向每个链表的头结点。（以同步激光雷达与相机图像为例）



同时，创建消息同步器时，第一个消息为Pivot，消息同步将会以该消息为基准进行。建议设置为频率最低的消息。

C++

```
std::vector<std::string> msg_names = {"/sensor/LiDAR", "/sensor/Image"};
std::vector<nalio::DataSyncer::SyncType> msg_sync_types = {
  nalio::DataSyncer::SyncType::kNearest,
  nalio::DataSyncer::SyncType::kNearest};
nalio::DataSyncer::Ptr lidar_img_syncer =
  buffer_.createDataSyncer(msg_names, msg_sync_types);
```

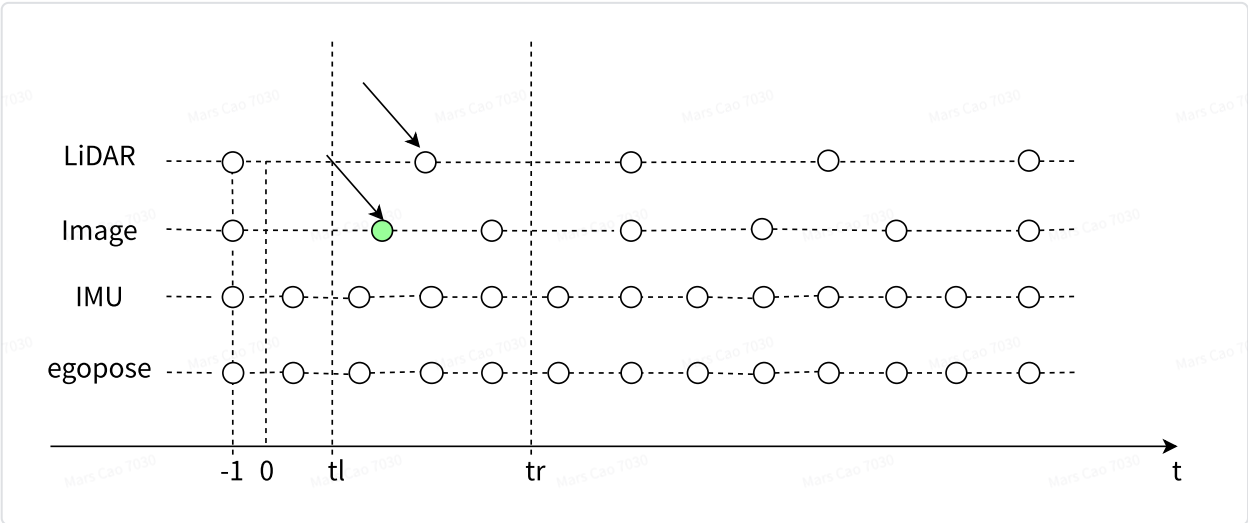
同步消息

Pivot 消息的同步类型不影响同步方式，其他消息的同步类型会影响同步方式，具体为

时间戳最近查找 Nearest

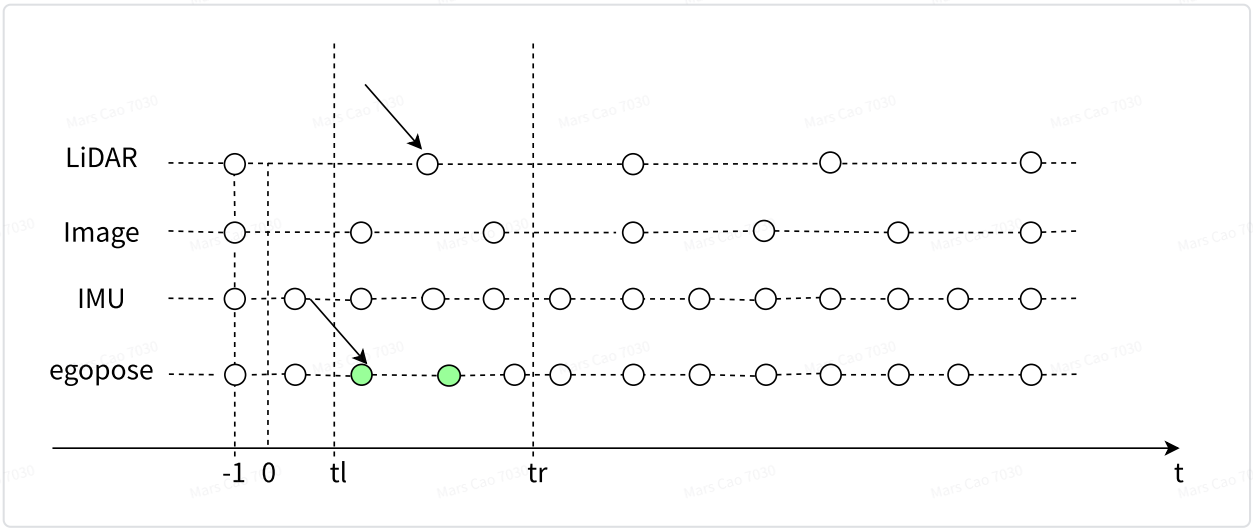
以同步激光雷达与图像为例，则根据Pivot 消息（激光雷达消息）查找最近的图像消息的方式为：

根据Pivot 迭代器指向的消息的时间戳，+- 阈值（通过Datasyncer::getSyncMessages设置），并将该类型的消息都取出，再求得时间戳最近的一个消息



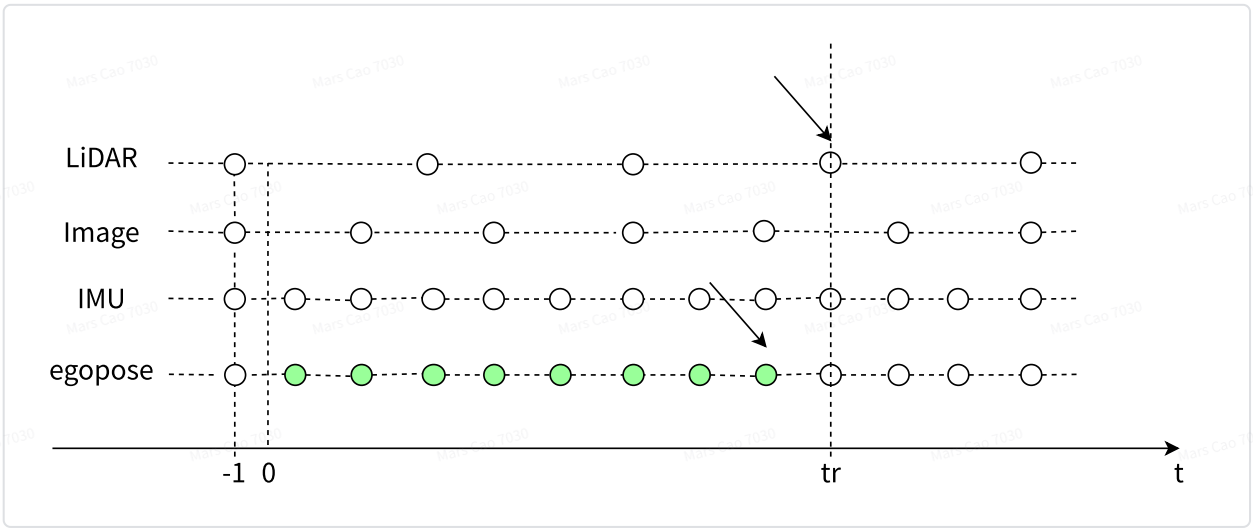
时间戳双向最近查找 Billateral

与上述方法类似，不同的是会返回Pivot timestamp左右最近的两个消息。通常在使用egopose 插值的场景下使用。



时间戳之前所有消息 Multiple

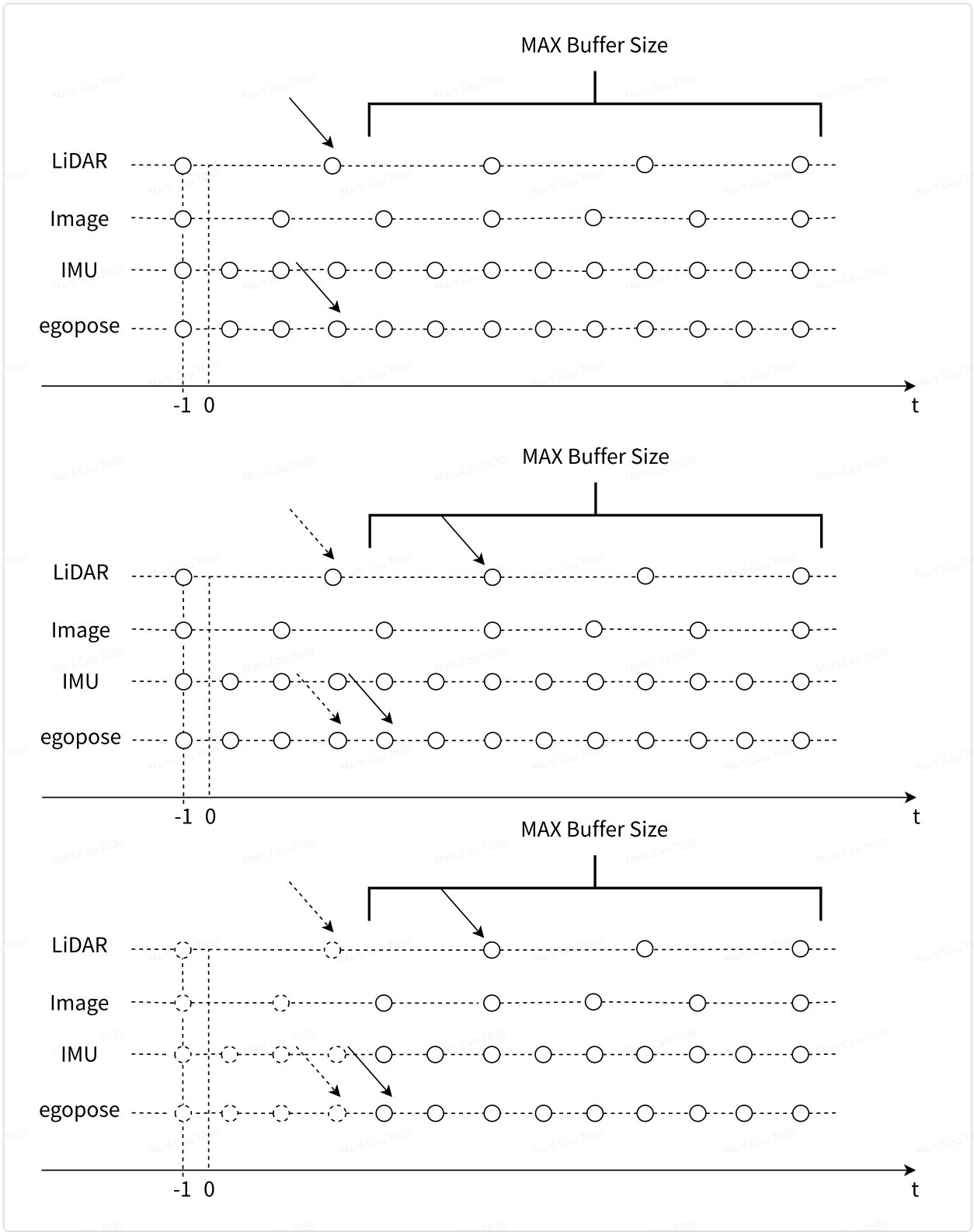
会将Pivot timestamp 之前的所有消息返回。通常用来将IMU的消息统统返回并在后续过程中做积分使用。



删除消息

DataBuffer 会在初始化时创建一个进程来周期性地查看每个消息队列是否超过设定的buff size。

如果超过了设定的buff size，并且有**同步器的迭代器**指向了待删除的消息，则会首先将所有的消息同步器的迭代器**向前推进**到待删除的最后一个节点之前的位置，再进行删除。



其他细节

基于atomic的读写锁设计

> **读写锁**是计算机程序的并发控制的一种同步机制，也称“共享-互斥锁”、多读者-单写者锁。[1]**多读者锁**，[2]，“push lock”[3]) 用于解决读写问题（英语：readers–writers problem）。读操作可并发重入，写操作是互斥的。

C++

```cpp
class ReadWriteMutex {
 public:
  ReadWriteMutex();

  void lockReader() {
    if (indicator_.load() < 0) {
      std::this_thread::yield();
    }
    ++indicator_;
  }
  void unlockReader() {
    --indicator_;
  }
  void lockWriter() {
    if (indicator_.load() != 0) {
      std::this_thread::yield();
    }
    indicator_.store(-1);
  }
  void unlockWriter() {
    indicator_.store(0);
  }

 private:
  std::atomic_int16_t indicator_;
};
```

只有当删除消息的时候，才会出现迭代器的失效的可能性。为此，在删除消息时进行写保护，而在其他操作迭代器的函数中进行读保护。