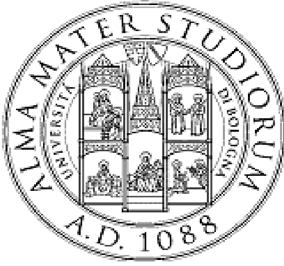


Università degli Studi di Bologna
Facoltà di Ingegneria

Corso di
Reti di Calcolatori M

DDS – Data Distribution Service

Luca Foschini

Anno accademico 2014/2015

Agenda

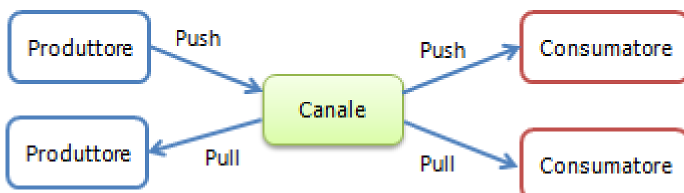
- Lo standard DDS
- Sviluppo di un applicazione DDS
- Content Subscription e QoS in DDS

Lo standard Data Distribution Service (DDS)

MIDDLEWARE: Modello molti a molti

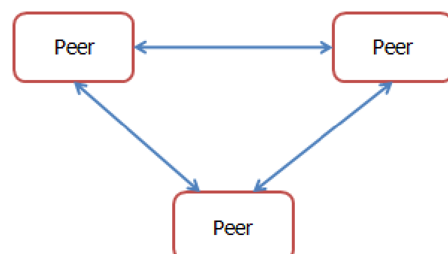
Due soluzioni principali:

Modelli centralizzati



Es. Corba Notification Service

Modelli peer-to-peer



Es. DDS

MIDDLEWARE: DDS

Standard OMG - Object Management Group

Paradigma: **Publish-Subscribe** *Topic-based*

Target: scenari distribuiti in cui è necessario scambiare dati in tempo reale da più sorgenti a più destinazioni

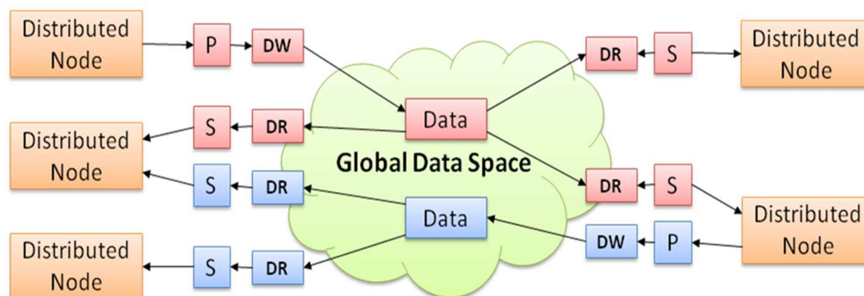
Ambiti di utilizzo: controllo industriale, difesa, trading finanziario, etc.

DDS: Caratteristiche

- **Disaccoppiamento tra entità:** nessuna conoscenza reciproca tra le entità prima dell'esecuzione
- **Qualità di Servizio (QoS):** altamente personalizzabile con possibile determinismo nella consegna delle informazioni, consegna realtime, persistenza messaggi e tolleranza ai guasti
- **Scalabilità:** capacità di adattarsi all'aumentare del carico
- **Dinamicità:** semplicità di aggiunta e rimozione nodi grazie all'accoppiamento lasco
- **Indipendenza dalla piattaforma / linguaggio**: interfacce astratte per i dati definite mediante IDL (come in CORBA)
- **Nessuna infrastruttura centralizzata:** comunicazione diretta tra publisher e subscriber

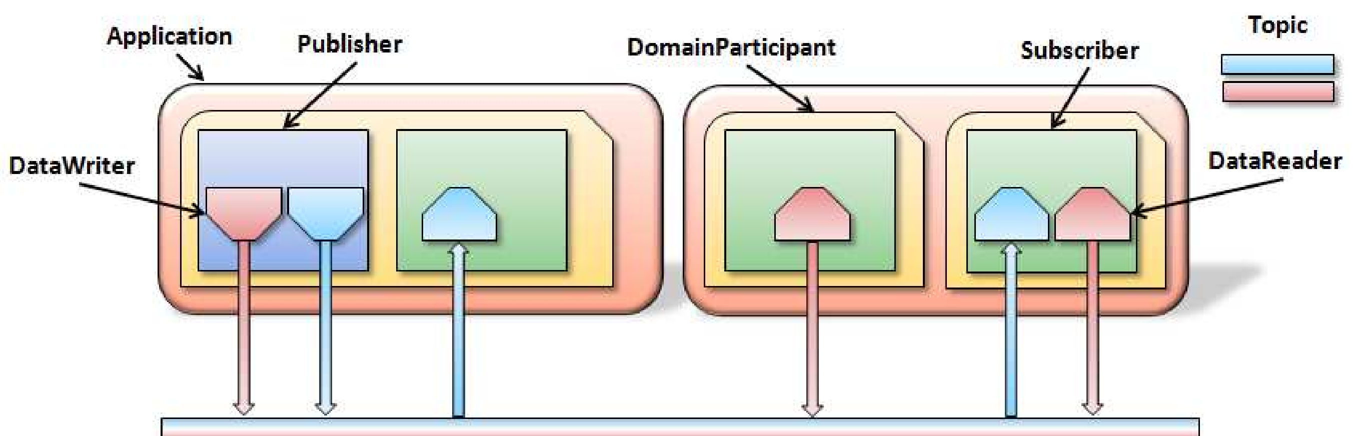
DDS: Entità

- **Publisher (P)**: entità che pubblica i dati (usando DataWriter)
- **Subscriber (S)**: entità che sottoscrive e riceve i dati (usando DataReader)
- **DataWriter (DW)**: end point che invia i dati su un particolare Topic
- **DataReader (DR)**: end point che riceve i dati su un particolare Topic
- **Dominio**: astrazione di spazio globale distribuito per i dati
- **DomainParticipant**: partecipante, punto di accesso al dominio (ogni applicazione ne ha almeno uno)



Ogni applicazione può avere più Publisher/Subriber
Ogni Publisher/Subscriber può avere più DataWriter/DataReader

DDS: Entità



- I DataWriter sono associati a tempo di compilazione ad un determinato Topic
- I DataReader sono associati a tempo di compilazione ad un determinato Topic, *ContentFilterTopic* o *MultiTopic*

I ContentFilterTopic e i MultiTopic sono dei particolari tipi di Topic che permettono di filtrare le informazioni in base al contenuto

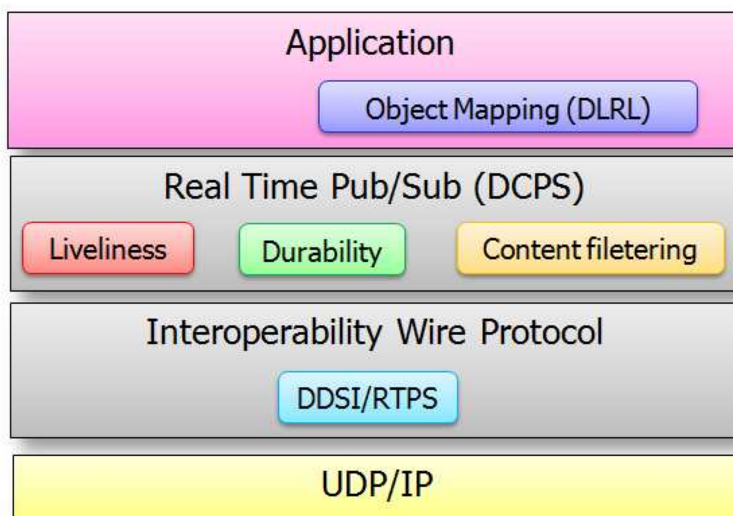
DDS: Architettura

- Tutte le parti dell'architettura sono distribuite sui nodi: nessun componente esterno/servizio di nomi/repository
- Discovery automatico a tempo di esecuzione: *Topic* e QoS come unici criteri di accordo tra publisher e subscriber

Data Local Reconstruction Layer (DLRL)
Vista object-oriented dei dati scambiati
Indipendente dall'infrastruttura

Data-Centric Publish Subscribe (DCPS)
Responsabile di distribuire i dati
Discovery automatico (plug & play) e fault-tolerance nella consegna dei dati
Diversi moduli di QoS configurabili

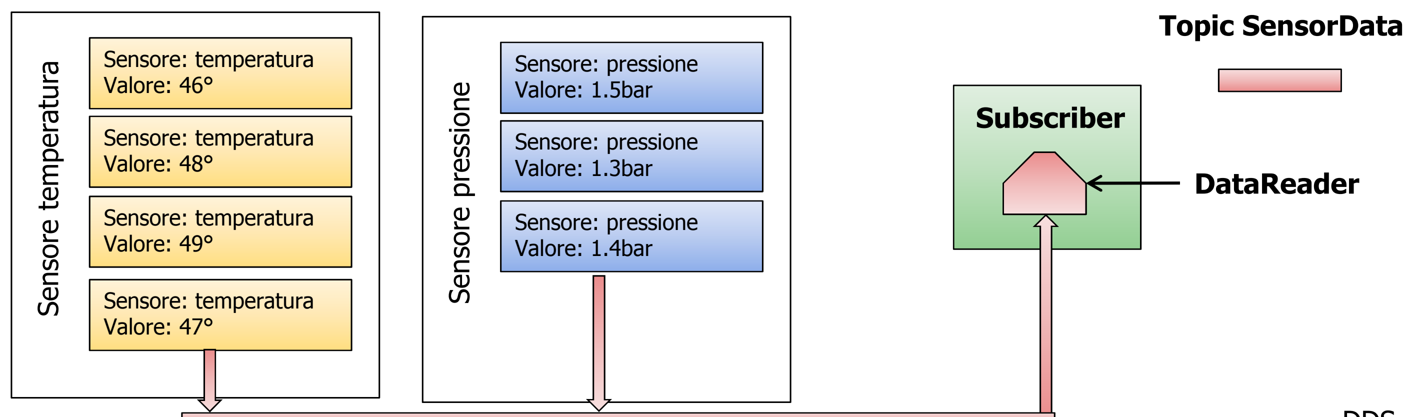
Interoperability wire protocol
Implementa lo strato di trasporto
Diversi protocolli utilizzabili: il più diffuso è Real-Time Publish/Subscribe (RTPS).
(***Multicast*** vs Unicast, discovery vs data transfer)



Content subscription: le chiavi

Le **chiavi** permettono di definire più istanze di uno stesso Topic, senza la necessità di creare più Topic per lo stesso tipo di dato

- Una chiave non è altro che **un insieme di attributi di un dato** che lo identifica univocamente come istanza. Ad esempio, unico topic per tutti i sensori, differenziati per key su campo "sensore" (possibilità di passare da un sensore all'altro senza cambiare topic)
- In DDS IDL è sufficiente marcare un campo con l'annotazione **@key**



Content subscription: filtri sui Topic

ContentFilteredTopics permette di filtrare i Topic in base al contenuto:

```
mittente = 'Corradi' and corso = 'Reti M'
pressione > 2 or temperatura > 100
```

- Possibile esprimere le espressioni in linguaggio simil-SQL
- Il middleware si preoccupa di filtrare i dati prima di passarli all'applicazione

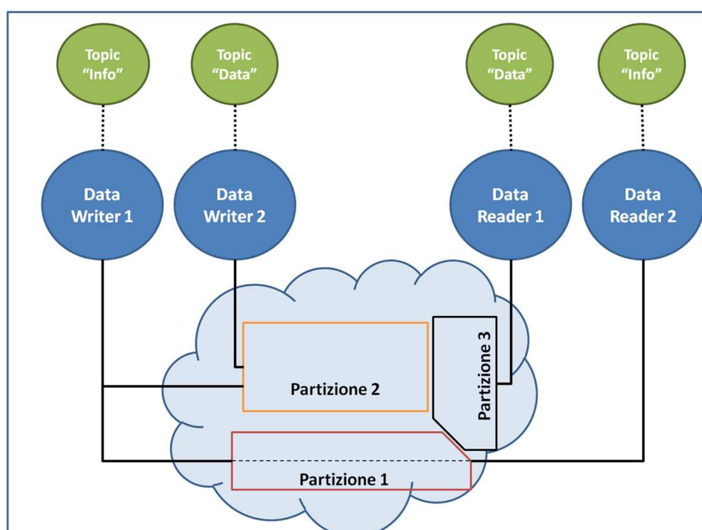
MultiTopic permette di sottoscrivere ***più Topic contemporaneamente***

- Il middleware si preoccupa di filtrare i dati e combinarli prima di passarli all'applicazione. È possibile per l'applicazione ricevere un tipo di dato diverso da quello del singolo Topic ottenuto mediante ***combinazione dei tipi dei Topic***
- Permette di ***astrarre il tipo di dato applicativo*** dai dati effettivamente trasmessi dai Publisher consentendo ai publisher di cambiare i tipi di dato trasmessi senza influenzare i subscriber. L'infrastruttura si preoccupa di ricostruire il dato applicativo dai nuovi tipi di dato

Content subscription: partizioni DDS

Le **partizioni** sono spazi di nomi che permettono di dividere logicamente un dominio DDS

I Publisher/Subscriber possono decidere **a tempo di esecuzione** (e non a tempo di creazione come i Topic) su quali partizioni pubblicare/sottoscrivere i dati



Affinché un DataReader possa ricevere i messaggi di un DataWriter è necessario condividere **sia lo stesso Topic sia la stessa partizione**

Le partizioni sono considerate come una policy di QoS

Guide linea e direzioni di implementazione

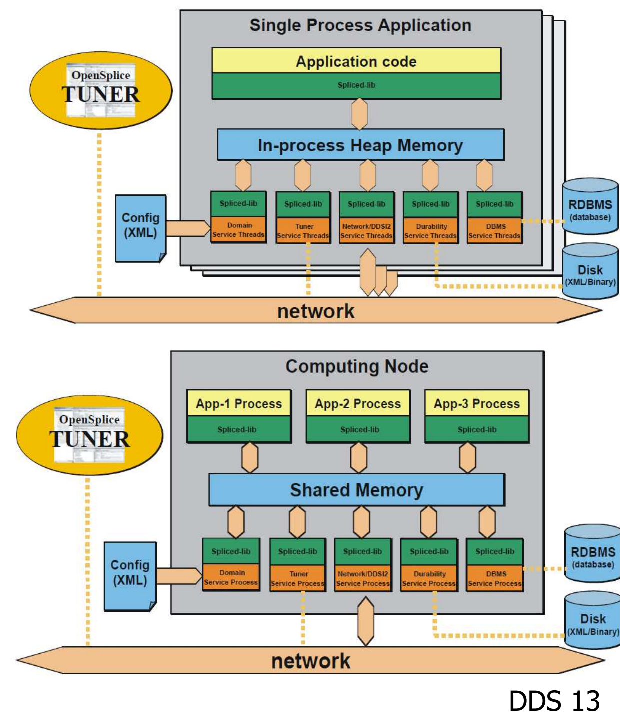
Le specifiche **DDS** permettono alle applicazioni **DCPS** di trarre vantaggio da diversi **modelli di comunicazione**: **unicast**, **multicast**, **broadcast**

Diversi **modelli architetturali distribuiti** per la distribuzione di dati

- **modello decentralizzato**: logica applicativa e supporto DCPS in esecuzione sullo **stesso** processo utente
- **modello federato**: logica applicativa e supporto DCPS in esecuzione su **diversi** processi utente

Modelli di ottimizzazione dello scambio di dati per migliorare le performance nella disseminazione di dati

- **data batch**: molteplici dati concentrati in un unico datagram packet



Implementazioni DDS: Prismtech OpenSplice

Implementazione basata su un'**architettura federata**

- separazione tra **logica applicativa**, configurazione **DCPS** e **dettagli di comunicazione**
- il demone **DCPS** potrebbe risultare un potenziale **collo di bottiglia** ad alte frequenze di invio dei dati
- il demone **DCPS** costituisce un potenziale **single point of failure**

OpenSplice esegue di default un'ottimizzazione dello scambio di dati basata sul **data batching.** Supporta, inoltre, modelli di comunicazione di tipo **unicast, multicast e broadcast.**

Supporta due diversi **protocolli di invio dati**

- protocollo proprietario **RTNetworking**
- protocollo standard **Real-Time Publish-Subscribe (RTPS)** per garantire l'interoperabilità con altre implementazioni DDS

A partire dalla versione 6, OpenSplice può essere utilizzato anche in modalità "**standalone**" secondo un **modello** architetturale di tipo **decentralizzato**

Implementazioni DDS: RTI Connext DDS

Implementazione basata su un'**architettura decentralizzata**

- le **applicazioni** sono **autocontenute**
- conseguente riduzione della **latenza**
- nessun **collo di bottiglia** o **single point of failure**

RTI di default non supporta nessuna ottimizzazione dello scambio di dati basata sul **data batching.** Supporta, invece, modelli di comunicazione di tipo **unicast e multicast.**

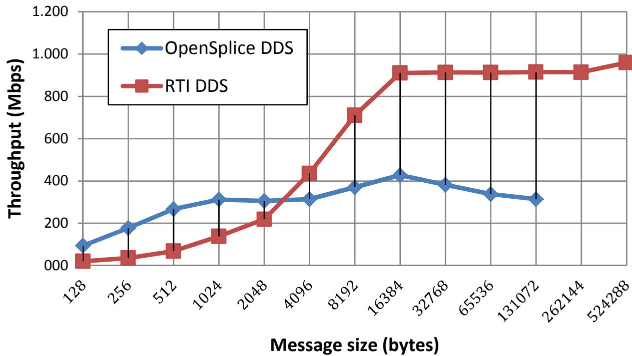
Supporta un solo **protocollo di invio dati**

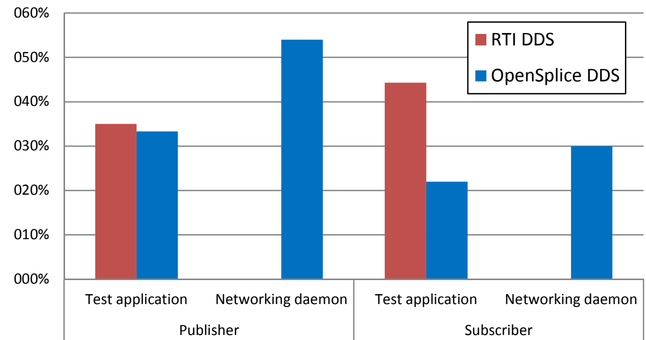
- protocollo standard **Real-Time Publish-Subscribe (RTPS)** per garantire l'interoperabilità con altre implementazioni DDS (es. OpenSplice)

Differenze tra OpenSplice e RTI

	Comunicazione	Architettura distribuita	Protocolli di invio dati	Ottimizzazioni (default)
OpenSplice	• unicast • multicast • broadcast	modello federato	• RTNetworking (default) • RTPS	• data batching
RTI	• unicast • multicast	modello decentralizzato	• RTPS	

RTI e OpenSplice: un confronto di prestazioni



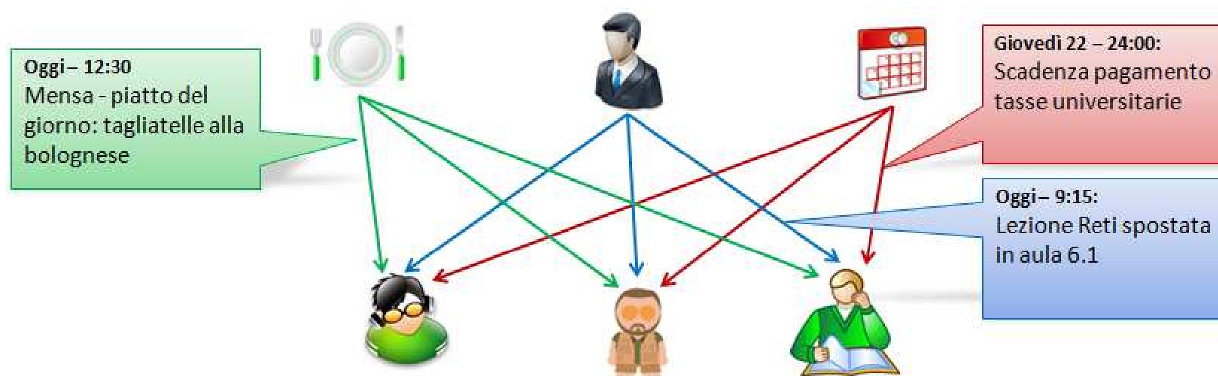


- **OpenSplice** risulta avere **performance migliori** rispetto a RTI per dati di **piccole dimensioni**
  - Il ***data batching*** permette l'invio di un grandi volumi di dati formati da piccoli pacchetti
- Con dati di **dimensione** più **grande**, a partire da 4 KB, la tendenza si inverte, con **RTI** che risulta più **performante** rispetto ad OpenSplice
- **RTI** riesce a **saturare l'intera banda** (1Gbps) con pacchetti di grandezza superiore a 16KB, mentre il **demone di Networking** di OpenSplice non riesce a gestire l'elevata produzione di dati e **fallisce** a 128KB

Sviluppo di un'applicazione in DDS

TimeService: UNIBO alarm service

L'università di Bologna ha deciso di sviluppare un servizio che in base all'ora della giornata permetta agli studenti di essere informati sugli eventi imminenti mediante l'invio di messaggi (tipo SMS)



Il servizio prevede **due applicazioni**: una per inviare le notifiche (**TimeService**) e una per riceverle (**TimeClient**)

È necessario anche definire un **formato dei dati comune** alle applicazioni/piattaforme

Passi di sviluppo di un'applicazione DDS

Modello di sviluppo basato sui dati

1. Definire un formato per i dati mediante IDL
2. Sviluppare i Publisher che diffondono le informazioni
3. Sviluppare i Subscriber che ricevono le informazioni

In scenari reali non sempre esiste una netta distinzione Publisher e Subscriber: spesso le applicazioni implementano entrambi i ruoli

TimeService: formato dei dati

Per i messaggi definiamo un tipo di dato **TimeMessage** indipendente dal linguaggio di implementazione

DDS IDL: sottoinsieme di IDL di Corba (non implementa tutti i costrutti)

Possibilità di definire strutture utilizzando solo tipi primitivi, stringhe e sequenze (liste/array)

Possibilità di raggruppare le strutture in moduli (spazi di nomi)

```
/*TimeMessage.idl
 *Struttura TimeMessage che contiene
 *un timestamp e un messaggio
 *time -> Data e ora dell'evento
 *message -> Descrizione dell'evento
 */
struct TimeMessage {
  long long time;
  string<256> message;
};
```

Partendo dall'IDL si possono generare automaticamente:

- Le classi Java che rappresentano il dato
- I DataWriter e i DataReader associati al tipo di dato
- Classi di supporto per lavorare con il dato

TimeService: implementazione

Per implementare il servizio di invio dei messaggi è sufficiente:

1. Creare un DomainParticipant mediante DomainParticipantFactory

```
DomainParticipant create_participant (int domainId, DomainParticipantQos
   qos, DomainParticipantListener listener, int mask)
```

   – Un Participant può avere dei listener per ricevere gli eventi non associati alle proprie entità; bit-mask mask per definire quali eventi gestire

2. Registrare il tipo di dato al DomainParticipant (ottenibile dalla classe autogenerata TimeMessageTypeSupport)

```
void register_type(DomainParticipant participant, String type_name)
```

3. Creare un Publisher dal DomainParticipant (opzionale – alternativa: creazione implicita quando si crea un DataWriter)

```
Publisher create_publisher (PublisherQos qos, PublisherListener listener,
int mask)
```

   – Un Publisher può avere dei listener per ricevere gli eventi non associati ai propri DataWriter; bit-mask mask definisce quali eventi gestire

TimeService: implementazione

4. Creare un Topic.

```
Topic create_topic (String topic name, String type_name, TopicQos qos,
TopicListener listener, int mask)
```

   – Un Topic può avere dei listener per ricevere gli eventi non associati agli specifici DataWriter/DataReader; bit-mask mask definisce quali eventi gestire

5. Creare un DataWriter dal DomainParticipant o dal Publisher per lo specifico tipo di Topic

```
DataWriter create_datawriter (Topic topic, DataWriterQos qos,
DataWriterListener listener, int mask)
```

   – Un DataWriter può avere dei listener per ricevere gli eventi sullo stato della comunicazione; bit-mask mask definisce quali eventi gestire

6. Implementare la logica di servizio di invio dei dati

TimeService: realizzazione Java

```java
public static void main(String[] args) {

    // 1. Creiamo un DomainParticipant sul dominio 0
    DomainParticipant participant = DomainParticipantFactory.get_instance()
        .create_participant(0, DomainParticipantFactory.PARTICIPANT_QOS_DEFAULT,
                null, StatusKind.STATUS_MASK_NONE);
    if (participant == null) {System.err.println("Impossibile creare un domain participant");return;}

    //2. Registriamo il tipo di dato
    TimeMessageTypeSupport.register_type(participant, TimeMessageTypeSupport.get_type_name());

    // 4. Creiamo un Topic di tipo TimeMessage
    Topic topic = participant.create_topic("TimeTopic", TimeMessageTypeSupport.get_type_name(),
        DomainParticipant.TOPIC_QOS_DEFAULT, null, StatusKind.STATUS_MASK_NONE);
    if (topic == null) {System.err.println("Impossibile creare un topic"); return;}

    // 5. Creiamo un datawriter (publisher implicito)
    TimeMessageDataWriter dataWriter = (TimeMessageDataWriter)
    participant.create_datawriter(topic,
        Publisher.DATAWRITER_QOS_DEFAULT, null, StatusKind.STATUS_MASK_NONE);
    if (dataWriter == null) {System.err.println("Impossibile creare un data writer"); return;}

    …
```

TimeService: realizzazione Java

```
    ...
    // 6. Implementazione della logica di servizio
    System.out.println("Sono pronto a inviare i messaggi (invio per continuare)");
    try { new InputStreamReader(System.in).read();} catch (IOException e) {}

    int i = 1;
    System.out.println("Inizio a inviare i messaggi");
    while (i<10)
    {
        TimeMessage message = new TimeMessage();
        message.time = new Date().getTime();
        message.message = "TimeService - message " + i;
        // Primitiva di scrittura ottenuta dal generatore IDL
        //void write(TimeMessage instance_data, InstanceHandle_t handle)
        //InstanceHandle_t utilizzato solo per ottimizzare le performance in casi di invio multiplo
        dataWriter.write(message, InstanceHandle_t.HANDLE_NIL);
        System.out.println("Messaggio "+i+" inviato");
        try { Thread.sleep(2000); } catch (InterruptedException e) {}
        i++;
    }

    participant.delete_contained_entities();
    DomainParticipantFactory.get_instance().delete_participant(participant);
}
```


TimeClient: implementazione

Per implementare il servizio di ricezione dei messaggi è sufficiente:

1. Creare un DomainParticipant mediante DomainParticipantFactory

   `DomainParticipant create_participant (int domainId, DomainParticipantQos qos, DomainParticipantListener listener, int mask)`

   – Un participant può avere dei listener per ricevere gli eventi non associati alle proprie entità; bit-mask mask definisce quali eventi gestire

2. Registrare il tipo di dato al DomainParticipant (ottenibile dalla classe autogenerata TimeMessageTypeSupport)

   `void register_type(DomainParticipant participant, String type_name)`

3. Creare un Subscriber dal DomainParticipant (opzionale – alternativa: creazione implicita quando si crea un DataReader)

   `Subscriber create_subscriber (SubscriberQos qos, SubscriberListener listener, int mask)`

   – Un subscriber può avere dei listener per ricevere gli eventi eventi non associati ai propri DataWriter; bit-mask mask definisce quali eventi gestire

TimeClient: implementazione

4. Creare un Topic

```
Topic create_topic (String topic name, String type_name, TopicQos qos,
TopicListener listener, int mask)
```

   – Un Topic può avere dei listener per ricevere gli eventi non associati agli specifici DataWriter/DataReader; bit-mask mask definisce quali eventi gestire

5. Creare un DataReader dal DomainParticipant o dal Subscriber per lo specifico tipo di Topic

```
DataReader create_datareader (Topic topic, DataReaderQos qos,
DataReaderListener listener, int mask)
```

   – Un DataReader può avere dei listener per ricevere gli eventi sullo stato della comunicazione (es. evento ricezione); bit-mask mask definisce quali eventi gestire

6. Implementare la logica di ricezione dei dati

TimeClient: realizzazione Java

```java
public static void main(String[] args) {

    // 1. Creiamo un DomainParticipant sul dominio 0
    DomainParticipant participant = DomainParticipantFactory.get_instance()
        .create_participant(0, DomainParticipantFactory.PARTICIPANT_QOS_DEFAULT,
                    null, StatusKind.STATUS_MASK_NONE);
    if (participant == null) {System.err.println("Impossibile creare un domain participant");return;}

    //2. Registriamo il tipo di dato
    TimeMessageTypeSupport.register_type(participant, TimeMessageTypeSupport.get_type_name());

    // 4. Creiamo un Topic di tipo TimeMessage
    Topic topic = participant.create_topic("TimeTopic", TimeMessageTypeSupport.get_type_name(),
        DomainParticipant.TOPIC_QOS_DEFAULT, null, StatusKind.STATUS_MASK_NONE);
    if (topic == null) {System.err.println("Impossibile creare un topic"); return;}

    // 5. Creiamo un datareader (subscriber implicito)
    TimeMessageDataReader dataReader = (TimeMessageDataReader) participant.create_datareader(topic,
        Subscriber.DATAREADER_QOS_DEFAULT, new TimeClient(), StatusKind.DATA_AVAILABLE_STATUS);
    if (dataReader == null) {System.err.println("Impossibile creare un data reader"); return;}

    …
```

TimeClient: realizzazione Java

```
    …
    // 6. Implementazione della logica di servizio
    System.out.println("Sono pronto a ricevere i messaggi (invio per terminare)");
    try { new InputStreamReader(System.in).read();} catch (IOException e) {}

    System.out.println("Attendere il rilascio delle risorse");
    participant.delete_contained_entities();
    DomainParticipantFactory.get_instance().delete_participant(participant);
}

@Override
public void on_data_available(DataReader dataReader) {
TimeMessageDataReader timeMessageDataReader = (TimeMessageDataReader) dataReader;
    SampleInfo info = new SampleInfo();
    for (;;) {
      try {
          TimeMessage message = new TimeMessage();
          timeMessageDataReader.take_next_sample(message, info);
          if (info.valid_data) {
            System.out.println("Ricevuto messaggio:" + message.message);
            System.out.println("Timestamp:" + new Date(message.time).toString() + "\n");
          }
          } catch (RETCODE_NO_DATA noData) { break;}
            catch (RETCODE_ERROR e) { e.printStackTrace(); }
    }
}
```


RTI DDS

- Implementazione commerciale di DDS (http://www.rti.com/)

INSTALLAZIONE SU LINUX

- Scaricare versione 4.4d per gcc4.1 (licenza 30 giorni o email a Luca Foschini)
- Definire il percorso di installazione: `export NDDSHOME=/percorso/ndds.4.4d_lic`
- Estrarre l'archivio in `/percorso` e copiare la licenza in `$(NDDSHOME)`

COMPILATORE IDL

```
rtiddsgen -language Java -package it.unibo.TimeMessage TimeMessage.idl
```

- Si trova in `$(NDDSHOME)/scripts`

ESEGUIRE LE APPLICAZIONI

- Esportare le librerie:

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$(NDDSHOME)/lib/i86Linux2.6gcc4.1.1jdk/
```

- Service:

```
java -cp .:$(NDDSHOME)/class/nddsjavad.jar it.unibo.TimeService.TimeService
```

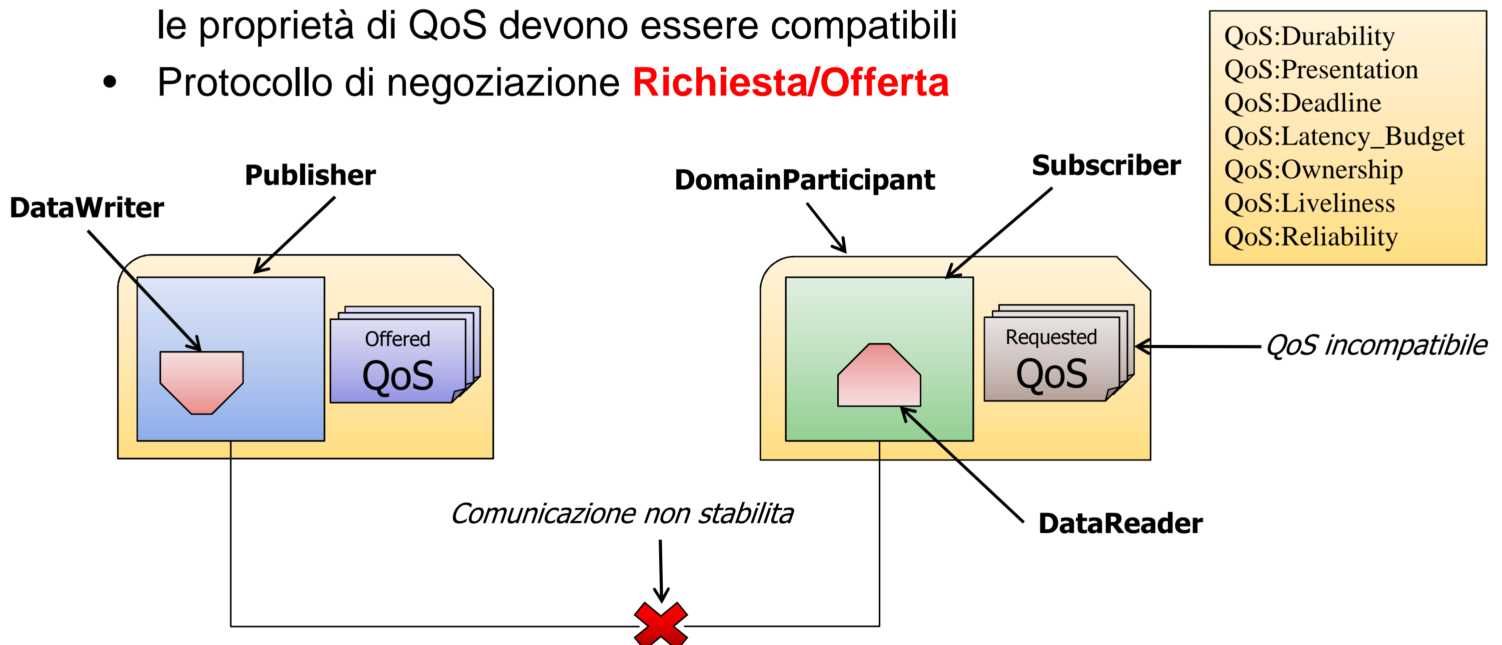
- Client:

```
java -cp .:$(NDDSHOME)/class/nddsjavad.jar it.unibo.TimeClient.TimeClient
```

Quality of Service in DDS

Quality of Service (QoS) in DDS

- Perché un Subscriber possa ricevere le pubblicazioni di un Publisher le proprietà di QoS devono essere compatibili
- Protocollo di negoziazione **Richiesta/Offerta**



DDS supporta diverse modalità di invio di messaggi (es. best-effort, reliable) e consente alle entità di gestire la persistenza delle informazioni

DDS QoS: reliability

DDS prevede due politiche di QoS per l'affidabilità dei messaggi: BEST_EFFORT e RELIABLE

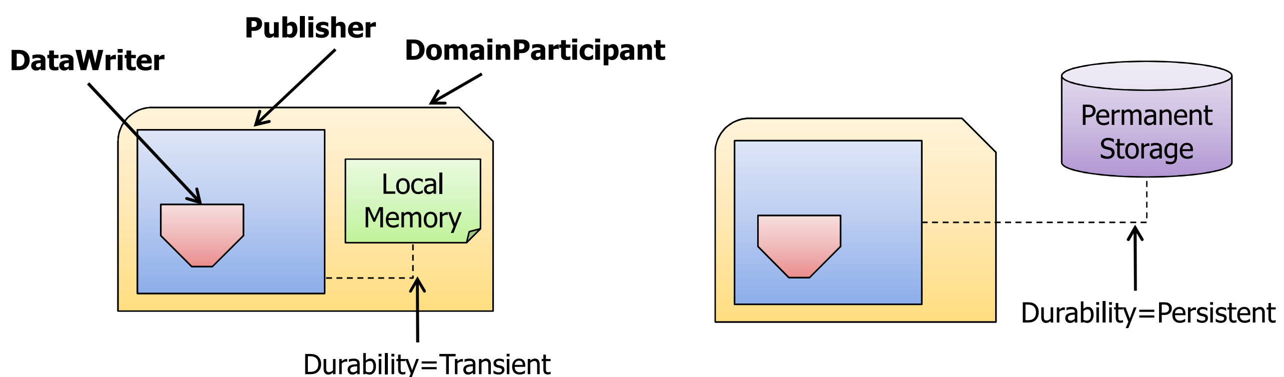
- Con la politica BEST_EFFORT non è garantito che tutti i messaggi siano ricevuti, né l'ordine di consegna
- Con la politica RELIABLE è garantito che tutti i messaggi siano ricevuti e l'ordine di consegna. In questo scenario i Publisher rinviano i dati ai Subscriber se necessario e i Subscriber danno un feedback (ack) ai Publisher quando ricevono i dati

Tutti i messaggi inviati sono mantenuti in una coda di history in attesa di essere confermati (lato publisher) e processati dall'applicazione (lato subscriber). La dimensione della coda è definibile mediante la policy di HISTORY

Possibile definire anche quante risorse (es. memoria, max istanze) utilizzare per il mantenimento dei dati mediante la policy RESOURCE_LIMITS

DDS QoS: durability

- Mediante la policy di Durability è possibile definire se e quanti dati mantenere lato publisher per poter essere richiesti successivamente.
- DDS supporta tre tipi di persistenza:
  - VOLATILE – No Instance History Saved
  - TRANSIENT – History Saved in Local Memory
  - PERSISTENT – History Saved in Permanent storage



DDS QoS: altre policy

DDS supporta una serie di altre policy per definire:

- Ordinamenti nella ricezione dei messaggi (DESTINATION_ORDER e PRESENTATION)
- Priorità dei messaggi (LATENCY_BUDGET)
- Esclusività su alcuni tipi di dato (OWNERSHIP)
- Autenticazione e sicurezza dei dati (USER_DATA)
- Vincoli temporali sul rate d'invio dei messaggi (TIME_BASED_FILTER)
- Fault detection e heartbeat (LIVELINESS)

Per la documentazione dettagliata si faccia riferimento a:

- Getting Started Guide: *www.rti.com/eval/rtidds44d/RTI_DDS_GettingStarted.pdf*
- RTI DDS User's Manual: *http://www.dre.vanderbilt.edu/~mxiong/tmp/backup/RTI_DDS_UsersManual.pdf*

Sviluppo di un'applicazione con QoS

- Estendiamo l'applicazione TimeService per supportare la QoS. Definiamo in modo esplicito i Publisher/Subscriber e specifichiamo le policy che vogliamo modificare

- La logica di servizio resta inalterata

TimeService: gestione QoS

```
public static void main(String[] args) {
    // 1, 2, 4. Creiamo un DomainParticipan, registriamo il tipo di dato, creiamo Topic
    ...
    // 3. Creiamo un Publisher esplicitamente
    Publisher publisher = participant.create_publisher(
        DomainParticipant.PUBLISHER_QOS_DEFAULT, null, StatusKind.STATUS_MASK_NONE);
    if (publisher == null) {System.err.println("Impossibile creare un publisher.");return; }

    // Definiamo le policy che vogliamo modificare
    DataWriterQos qos = new DataWriterQos();
    publisher.get_default_datawriter_qos(qos);
    qos.reliability.kind = ReliabilityQosPolicyKind.RELIABLE_RELIABILITY_QOS;
    qos.history.kind = HistoryQosPolicyKind.KEEP_LAST_HISTORY_QOS;
    qos.history.depth = 5;
    qos.durability.kind = DurabilityQosPolicyKind.TRANSIENT_LOCAL_DURABILITY_QOS;

    // 5. Creiamo un datawriter con le policy di QoS associate
    TimeMessageDataWriter dataWriter = (TimeMessageDataWriter) participant.create_datawriter(topic,
            qos, null, StatusKind.STATUS_MASK_NONE);
    if (dataWriter == null) {System.err.println("Impossibile creare un data writer"); return;}
    ...
    // 6. Implementazione della logica di servizio
    ...
}
```


TimeClient: gestione QoS

```
public static void main(String[] args) {
    // 1, 2, 4. Creiamo un DomainParticipant, registriamo il tipo di dato, creiamo un Topic
    ...
    // 3. Creiamo un subscriber esplicitamente
    Subscriber subscriber = participant.create_subscriber(
            DomainParticipant.SUBSCRIBER_QOS_DEFAULT, null, StatusKind.STATUS_MASK_NONE);
    if (subscriber == null) {System.err.println("Unable to create subscriber."); return; }

    // Definiamo le policy che vogliamo modificare
    DataReaderQos qos = new DataReaderQos();
    subscriber.get_default_datareader_qos(qos);
    qos.reliability.kind = ReliabilityQosPolicyKind.RELIABLE_RELIABILITY_QOS;
    qos.durability.kind = DurabilityQosPolicyKind.TRANSIENT_LOCAL_DURABILITY_QOS;

    // 5. Creiamo un datareader
    TimeMessageDataReader dataReader = (TimeMessageDataReader) participant.create_datareader(topic,
            qos, new TimeClient(), StatusKind.DATA_AVAILABLE_STATUS);
    if (dataReader == null) {System.err.println("Impossibile creare un data reader"); return;}

    ...
    // 6. Implementiamo la logica di servizio
    ...
}
```