UDI Developer Guide (release 0.2)

Rafael Oliveira Vasconcelos

LAC/ PUC-Rio

rvasconcelos@inf.puc-rio.br

Wrappers do SDDL

Um wrapper representa um tópico que é manipulado pelos serviços do SDDL. O wrapper permite que aplicações baseadas em SDDL sejam independentes de implementações específicas do padrão DDS (e.g. CoreDX, RTI Connext, OpenSplice), da mesma forma como objetos de aplicação permitem uma independência do Banco de Dados utilizado em uma aplicação.

Cada wrapper contém a lógica que converte o objeto em um tópico DDS para uma implementação específica. Atualmente, temos wrappers para os produtos DDS CoreDX e RTI Connext, mas pode-se implementá-los também para outros produtos DDS.

Para criar um novo wrapper, correspondente a um novo tópico no SDDL, a classe deve implementar a interface `ApplicationObject`.

Implementando um wrapper

Como exemplo, veja a implementação do wrapper `ConnectionReport`, que possui os ids do MN e do GW, além de um valor booleano para informar se o MN está conectado ou não.

```
public class ConnectionReport implements ApplicationObject {

  private UUID    nodeId;
  private UUID    gatewayId;
  private boolean connected;

  public ConnectionReport() {
  }

  public ConnectionReport(UUID nodeId, UUID gatewayId, boolean
connected) {
    this.nodeId = nodeId;
    this.gatewayId = gatewayId;
    this.connected = connected;
  }

  public ConnectionReport(ConnectionReportTopic
connectionReportTopic) {
    this.nodeId = new
UUID(connectionReportTopic.mostSignificantBitsVehicleId,
connectionReportTopic.leastSignificantBitsVehicleId);
    this.gatewayId = new
UUID(connectionReportTopic.mostSignificantBitsGatewayId,
connectionReportTopic.leastSignificantBitsGatewayId);
    this.connected = connectionReportTopic.connected;
  }
```

```
  public
ConnectionReport(lac.cnet.sddl.topics.rti.ConnectionReportTopic
connectionReportTopic) {
    this.nodeId = new
UUID(connectionReportTopic.mostSignificantBitsVehicleId,
connectionReportTopic.leastSignificantBitsVehicleId);
    this.gatewayId = new
UUID(connectionReportTopic.mostSignificantBitsGatewayId,
connectionReportTopic.leastSignificantBitsGatewayId);
    this.connected = connectionReportTopic.connected;
  }

  @Override
  public Object getSpecificDdsTopic(SupportedDDSVendors ddsVendor) {
    if (this.nodeId == null) {
      this.nodeId = new UUID(0, 0);
    }

    if (this.gatewayId == null) {
      this.gatewayId = new UUID(0, 0);
    }

    switch (ddsVendor) {
      case CoreDX:
        return this.getCoreDxTopic();
      case RTI:
        return this.getRtiTopic();
      default:
        return null;
    }
  }

  @Override
  public String getTopicTypeName(SupportedDDSVendors ddsVendor) {
    switch (ddsVendor) {
      case CoreDX:
        return ConnectionReportTopic.class.getCanonicalName();
      case RTI:
        return
lac.cnet.sddl.topics.rti.ConnectionReportTopic.class.getCanonicalName
();
      default:
        return null;
    }
  }

  ConnectionReportTopic getCoreDxTopic() {
    ConnectionReportTopic connectionReportTopic = new
ConnectionReportTopic();

    connectionReportTopic.connected = this.connected;
    connectionReportTopic.leastSignificantBitsGatewayId =
this.gatewayId.getLeastSignificantBits();
    connectionReportTopic.leastSignificantBitsVehicleId =
this.nodeId.getLeastSignificantBits();
    connectionReportTopic.mostSignificantBitsGatewayId =
this.gatewayId.getMostSignificantBits();
    connectionReportTopic.mostSignificantBitsVehicleId =
this.nodeId.getMostSignificantBits();

    return connectionReportTopic;
```

```
  }

  lac.cnet.sddl.topics.rti.ConnectionReportTopic getRtiTopic() {
    lac.cnet.sddl.topics.rti.ConnectionReportTopic
connectionReportTopic = new
lac.cnet.sddl.topics.rti.ConnectionReportTopic();

    connectionReportTopic.connected = this.connected;
    connectionReportTopic.leastSignificantBitsGatewayId =
this.gatewayId.getLeastSignificantBits();
    connectionReportTopic.leastSignificantBitsVehicleId =
this.nodeId.getLeastSignificantBits();
    connectionReportTopic.mostSignificantBitsGatewayId =
this.gatewayId.getMostSignificantBits();
    connectionReportTopic.mostSignificantBitsVehicleId =
this.nodeId.getMostSignificantBits();

    return connectionReportTopic;
  }
```

Exemplo de uso do wrapper por um serviço do SDDL

No exemplo abaixo são criados um participante, um Publisher, um Subscriber, o tópico *Message* e um Data Reader para este tópico.

```
    this.dds = UniversalDDSLayerFactory.getInstance();
    this.dds.createParticipant(UniversalDDSLayerFactory.CNET_DOMAIN);
    this.dds.createPublisher();
    this.dds.createSubscriber();

    GroupDefinerListener groupDefinerDDSListener = new
GroupDefinerListener(this);

    Object messageTopic = this.dds.createTopic(Message.class,
Message.class.getSimpleName());

    this.dds.createDataReader(groupDdefinerDDSListener, messageTopic);
```

O listener do exemplo acima recebe objetos do tipo *Message*, que é um wrapper:

```
public class GroupDefinerListener implements
UDIDataReaderListener<ApplicationObject> {

   /**
   * {@inheritDoc}
   */
  @Override
  public void onNewData(ApplicationObject nodeMessage) {
    //do something…

  }

}
```

DDS RTI

O RTI Connext, diferentemente do CoreDX DDS, possui apenas vetores de tamanho fixo, mesmo que na definição do tópico seja informado um vetor sem

tamanho fixo (neste caso o RTI usará o valor default). Desta forma, é necessário especificar o tamanho máximo de cada vetor, para evitar que o RTI crie os vetores com tamanho de 100 posições, que é o default do RTI.

Esta alteração pode ser feita na própria IDL, ou então alterando o valor default do tamanho máximo dos vetores do RTI.

Para informar um valor fixo do vetor já na definição do tópico, deve-se além do tipo do vetor informar o seu tamanho, como no exemplo abaixo:

```
struct ShortSequence {
    sequence<short, 200> short_sequence;
};
```

Para alterar o valor default, pode ser passado o parâmetro "-sequenceSize <TAMANHO>" no momento da compilação dos tópicos, como no exemplo abaixo:

CALL "%NDDSHOME%\scripts\rtiddsgen.bat" -package lac.cnet.sddl.topics.rti ***-sequenceSize 2000000*** *-language Java -ppDisable -replace SystemTopics.idl*

Perfis de QoS

A UDI permite a passagem de perfis de QoS para a criação dos Data Readers e Data Writers. Além disso, a UDI permite também que o perfil seja salvo/lido no formato JSON.

Atualmente, as políticas suportadas por ambas as entidades são:

- `DurabilityQosPolicy`
- `DeadlineQosPolicy`
- `LatencyBudgetQosPolicy`
- `LivelinessQosPolicy`
- `ReliabilityQosPolicy`
- `DestinationOrderQosPolicy`
- `HistoryQosPolicy`
- `ResourceLimitsQosPolicy`
- `LifespanQosPolicy`
- `OwnershipQosPolicy`

Além destas políticas, é possível "setar" também as seguintes políticas para o Data Reader:

- `ReaderDataLifecycleQosPolicy`
- `TimeBasedFilterQosPolicy`

Por fim, o Data Writer possui também a seguintes políticas:

- `DurabilityServiceQosPolicy`
- `TransportPriorityQosPolicy`
- `OwnershipStrengthQosPolicy`

- WriterDataLifecycleQosPolicy
- BatchQosPolicy

Exemplo de Como Setar um Perfil de QoS

```
    QosProfile qosProfile = new QosProfile();
    qosProfile.dataReaderQos = new DataReaderQos();
    qosProfile.dataWriterQos = new DataWriterQos();

    qosProfile.dataReaderQos.reliability = new ReliabilityQosPolicy();
    qosProfile.dataReaderQos.reliability.kind =
ReliabilityQosPolicyKind.BEST_EFFORT_RELIABILITY_QOS;

    qosProfile.dataWriterQos.reliability = new ReliabilityQosPolicy();
    qosProfile.dataWriterQos.reliability.kind                              =
ReliabilityQosPolicyKind.BEST_EFFORT_RELIABILITY_QOS;
```

Para o exemplo acima, seria criado o seguinte arquivo JSON:

```
{"dataReaderQos":{"reliability":{"kind":"BEST_EFFORT_RELIABILITY_QOS"}},"da
taWriterQos":{"reliability":{"kind":"BEST_EFFORT_RELIABILITY_QOS"}}}
```

Exemplo de Uso da API

Dado que o perfil de QoS já foi criado, ele pode ser salvo no formato JSON do seguinte modo:

```
udi.writeQosProfileToJsonFile(qosProfile, “qosProfile.txt”);
```

Já para criar um Data Reader com QoS utilizando um objeto DataReaderQos:

```
udi.createDataReader(applicationDataReaderListener,       topicDescription,
dataReaderQos)
```

Ou alternativamente, é possível criar o Data Reader com QoS a partir de um arquivo JSON armazenado no disco, para tal o código é:

```
udi.createDataReader(applicationDataReaderListener,       topicDescription,
“qosProfile.txt”)
```

O Data Writer pode ser criado de maneira análoga:

```
udi.createDataWriter(topicDescription, dataWriterQos)
```

Ou alternativamente:

```
udi.createDataWriter(topicDescription, “qosProfile.txt”)
```

Exemplo Simples

No exemplo abaixo são criados um participante, um Publisher, um Subscriber, o tópico *Message*, um Data Reader e um Data Writer para este tópico.

```
    this.dds = UniversalDDSLayerFactory.getInstance();
    this.dds.createParticipant(UniversalDDSLayerFactory.CNET_DOMAIN);
    this.dds.createPublisher();
    this.dds.createSubscriber();
```

```
    GroupDefinerListener groupDefinerDDSListener = new
GroupDefinerListener(this);

    Object messageTopic = this.dds.createTopic(Message.class,
Message.class.getSimpleName());

    this.dds.createDataReader(groupDefinerDDSListener, messageTopic,
“qosProfile.txt”);

    this.dds.createDataWriter(messageTopic, “qosProfile.txt”);
```

Sobre as Políticas de QoS

Esta seção explica brevemente como funciona cada política de QoS suportada pela UDI e apresenta possíveis casos de uso.

- **DurabilityQosPolicy:** Especifica se o DDS deve armazenar os dados publicados para que um novo DataReader possa receber os dados já publicados. Esta política pode ser útil quando se deseja receber todo o histórico de mensagens já enviadas ou por questões de tolerância a falha. Por exemplo, caso um GW falhe, ao ser iniciado novamente pode receber as mensagens que foram enviadas para ele no momento em que estava off-line.
- **DeadlineQosPolicy:** No DR, é utilizado para especificar o tempo máximo esperado para receber um dados. No DW, é utilizado para especificar o tempo máximo entre 2 publicações. É útil para quando se deseja explicitar o tempo entre as publicações e assim poder detectar caso o desempenho do sistema esteja degradado.
- **LatencyBudgetQosPolicy:** Informa quanto tempo o DDS pode esperar antes de fazer o envio do dado. Útil para priorizar alguns tópicos e permitir o batch dos dados, reduzindo assim o custo de rede.
- **LivelinessQosPolicy:** Permite ao DR identificar quando não há mais uma conexão com o DW, ou seja, com o DW perdeu conexão ou falhou. É útil para verificar que a “conexão” entre o DR e o DW está funcional. Em outras palavras, se o DW está ativo.
- **ReliabilityQosPolicy:** Informa se o envio dos dados deve ser feito de modo confiável ou não. Pode-se escolher entre melhor esforço ou confiável. O envio de dados periódicos, como por exemplo o envio da carga de trabalho do GW, pode ser enviado de maneira não confiável, já que um novo dado será escrito em breve.
- **DestinationOrderQosPolicy:** Controla se os dados precisam ser entregues em ordem ou não.
- **HistoryQosPolicy:** Informa quantos dados podem ser armazenados para a comunicação confiável ou serviço de durabilidade. Caso o histórico seja mantido com valor 5, por exemplo, caso 6 dados sejam escritos e por algum motivo tenham sido “dropados” na rede, 1 deles será perdido, pois o histórico só armazena os últimos 5 dados. O histórico também afeta a quantidade de dados que serão armazenados no serviço de durabilidade.
- **ResourceLimitsQosPolicy:** Informa o máximo de memória que o DR/DW pode alocar. Útil para sistemas embarcados que precisam explicitar o

máximo de memória que podem utilizar. Além disso, alocação dinâmica de memória pode causar latências não-determinísticas.

- **LifespanQosPolicy:** Especifica o atraso máximo entre a escrita e a leitura para que o dado seja considerado como válido pelo DR. Considerando um dado que é escrito a cada 1 s, um dado recebido depois de 1s pode ser considerado como inválido.
- **OwnershipQosPolicy:** Especifica se um DR pode receber dados de múltiplos DWs, comportamento padrão do DDS. Esta política é útil quando se tem réplicas de um determinado serviço, entretanto o DR deve receber dados apenas da instância ativa, aquela considerada "master". Normalmente utilizada em conjunto com Liveliness e Deadline.
- **ReaderDataLifecycleQosPolicy:** Controla se o DR deve remover do cache os dados recebidos. Por padrão o listener da UDI obtém e apaga os dados do cache do DR.
- **TimeBasedFilterQosPolicy:** Informa o tempo mínimo entre o recebimento dos dados no DR. Considerando que o GW envia a sua carga de trabalho a cada 10s e uma aplicação só está interessada em receber esta informação a cada 30s, o DR deve configurar está política para 10s.
- **DurabilityServiceQosPolicy:** Quando utilizada a política de durabilidade, está política é útil para configurar o serviço que fará a persistência.
- **TransportPriorityQosPolicy:** Informa ao DDS que determinado tópico tem prioridade em detrimento a outros. Esta informação é repassada para o protocolo e sistema operacional utilizado pelo DDS caso eles tenham suporte. Alguns tópicos, como o Pingo, por exemplo, pode ter prioridade maior do que outros tópicos.
- **OwnershipStrengthQosPolicy:** Informa a prioridade que o DW tem para escrever um dado quando se está utilizando a política de Ownership. O DW com o maior valor será considerado o que tem exclusividade para escreve os dados.