The Architecture and Implementation of the BacGrid Simulator

Michael Lees Brian Logan John King

May 2006

1 Introduction

In this working paper, we describe the architecture and implementation of BACGRID, a prototype Grid-based simulator for bacterial biofilms. Three versions of BACGRID have been implemented: a non-distributed version, a version based on the High Level Architecture (HLA) simulator interoperability framework which allows the distribution of simulation components across multiple processors in a single cluster or at remote sites, and a Grid version which allows distribution of simulation components on the Grid. In the next section, we briefly review the biofilm model on which BACGRID is based. In section 3 we outline the architecture of the simulator, before going on the describe the various versions of the simulator in detail in sections 4 to 6.

2 Biofilm Model

The BACGRID simulator implements the biofilm model described in [3]. For convenience, we summarise the key features of the biofilm model below.

The model system comprises a simple 3D 'biofilm reactor' consisting of two compartments, bulk liquid and biofilm. The bulk liquid compartment contains a (well mixed) solution of S different soluble substrates. The biofilm grows in a rectangular box of dimensions L_X, L_Y, L_Z with periodic x and y boundaries, and is assumed to consist of B different types of biomass. In addition to the biomass itself, the biofilm compartment contains a single type of extracellular polysaccharide (EPS) and Q different types of quorum sensing molecule. The biofilm and bulk liquid compartments are in contact and exchange solutes only by diffusion. Substrate and biomass which move beyond the x and y boundaries reappear at the opposite boundary. Bacteria, substrates and other material are assumed to be washed away once they reach the z boundary (detachment layer). For efficiency of computation, individual cells are aggregated into bacterial 'particles', as in [8]. Each particle represents a variable number of cells of a single bacterial strain. Particles allow the use of aggregated models of continuous processes (growth, division and displacement); however some processes must be modelled at the level of individual cells. Cells within a particle can exist in two different states: up-regulated and down-regulated. Particles keep track of the number of up-regulated and down-regulated cells they currently contain and cells can change from one state to another at each timestep, in response to the level of QSMs.

The biofilm compartment is discretised into sub-compartments or 'voxels' containing substrate and signalling molecules. Substrate and QSM concentrations are assumed

to be uniform across each individual voxel, and the upper bound on the size of a voxel is chosen such that the substrate and QSM concentration values are 'reasonably close' to the continuous values. The size of voxels, l_X, is chosen appropriately for the system to be modelled, with smaller values (criteria being deduced from the corresponding continuum models) giving greater resolution at increased computational and communication cost. However the voxels are typically fairly large in relation to the size of a cell, e.g., each voxel may contain of order 10^2 particles/10^4 cells. Each voxel contains zero or more particles of each biomass type (including EPS). The particles in a voxel exert a 'pressure' on the particles in the neighbouring voxels which is a function of the relative number of particles in the voxels, and these pressures are used to displace particles during the division of biomass. The arguments used in developing this pressure model are again based on the continuum modelling, in this case building on multiphase formulations for growing populations such as those described in [1]. Each voxel has six adjacent voxels, connected at each face, which are considered in determining relative pressures, and into which particles may be displaced. Voxels have a pre-determined maximum particle capacity, N, and the pressure in the voxel is considered to be infinite when this maximum is reached. N is calculated using l_X and the maximum radius of a particle, R, assuming simple cubic packing. EPS particles behave in the same way as biomass particles for the purposes of the pressure calculation. Each particle has a notional 3D position within its containing voxel which is used for visualisation purposes (see Figure 1). These notional positions are chosen such that the particles do not overlap. The pressure model and maximum particle size are chosen to ensure that there is enough free space in the voxel for this to be possible.

There are two main processes which determine the evolution of the model: the diffusion of substrate and QSM from voxel to voxel and changes in state of the particles in response to the substrate and QSM concentrations in their surrounding voxel. Particles are modelled as agents and implement a simple model of growth, division and displacement similar to that in [4], and up-regulation (e.g., the production of extracellular polysaccharide) in the presence of QSM [10]. These two processes interact: particles consume substrate and produce QSM, leading to transport associated with the diffusion gradients. The transport between voxels corresponds precisely to a simple central-difference discretisation of the relevant continuum reaction-diffusion equations.

The model provides a generic multi-scale framework for modelling populations of cells, which spans from the cellular level to the population level. In contrast to previous work, e.g., [8], it incorporates both aggregated and individual models of cellular processes, allowing the resolution of the model to be tailored for a particular modelling problem, while at the same time remaining computationally tractable. For example, in experiments to investigate the effect of QSM inhibitor on the up-regulation of the population, we have successfully simulated models containing 10^6 cells (10^4 particles) on a single processor. While the cell models used in the current prototype are naturally somewhat simplistic, the approach provides a generic framework into which different types of cells and more detailed models of gene expression and signalling can be plugged.

3 System Architecture

In this section we describe the high level architecture of the BACGRID simulator.

BACGRID is intended as a framework for systems biology simulations. The development of such simulations typically requires collaborative effort from researchers

with different domain knowledge and expertise, often at different locations. Support for collaborative model development and distributed execution of the resulting simulation models was therefore a key objective in the design of BACGRID.

To support collaborative model development, BACGRID utilises a combination of two emerging standards and their supporting middleware: the Grid and the High Level Architecture. The Grid supports e-Science through resource discovery, secure access to remote computational resources, data archiving and sharing etc., allowing virtual research teams to collaborate to solve research problems. The High Level Architecture (HLA) is an IEEE standard [2] for simulator interoperability, which supports the creation of distributed, composable simulations. These two technologies are complementary, and in combination they offer the promise of "on-demand" development of systems biology simulations. The interoperability of simulators provided by HLA is critical to the collaborative development of biological simulations and effective reuse of simulation components. In addition, many of the services necessary to support dynamic composition of simulations, e.g., model discovery and matching, secure execution, migration and load balancing, sharing and archival of simulation results etc., which are not addressed by the HLA standard, can potentially be provided by the Grid infrastructure.

To facilitate inter-operation and distribution, the implementation of the BACGRID simulation model is decomposed into a 'diffusion module' and one or more 'model regions'. The diffusion module is responsible for diffusing substrate and signalling molecules throughout the entire computational domain. Each model region processes one or more voxels, and handles the growth and division of particles within voxels, and the displacement of particles between voxels. In addition there is a visualisation module for run-time monitoring of the simulation progress and post-simulation analysis of results (see Figure 1). The model region and visualisation modules are implemented using the MASON agent toolkit [6].[1] (The visualisation module does not use the simulation capabilities of MASON, but makes extensive use of MASON's 3D libraries.) The diffusion module is written in Java.

Interaction between modules can be by means of procedure calls, HLA [2] (RTI) calls, or Grid invocations, each resulting in a different version of the core BACGRID system. The non-distributed (procedure call) version was used for initial model validation and as a benchmark for the evaluation of the distributed versions. The HLA version allows the distribution of BACGRID modules across multiple processors in a single cluster or at remote sites. The HLA distribution uses the DMSO RTI 1.3NGv6 Java bindings from the DMSO RTI reference distribution. The Grid version extends the HLA version to allow BACGRID modules to be distributed as Grid services. The Grid distribution is based on HLA_GRID [12], which allows HLA-compliant simulators to be instantiated and linked using Grid services.

In what follows, we discuss the three versions of BACGRID in turn, explaining how each version builds on its predecessor, and how distribution and inter-operation are achieved in the HLA and Grid versions.

4 Non-distributed Version

In this section we describe the non-distributed version of BACGRID in which the modules communicate via procedure calls. For simplicity, we assume that each model

[1] Some minor extensions to mason were required to support multi-phase `Steppable` objects and extendable sequences.

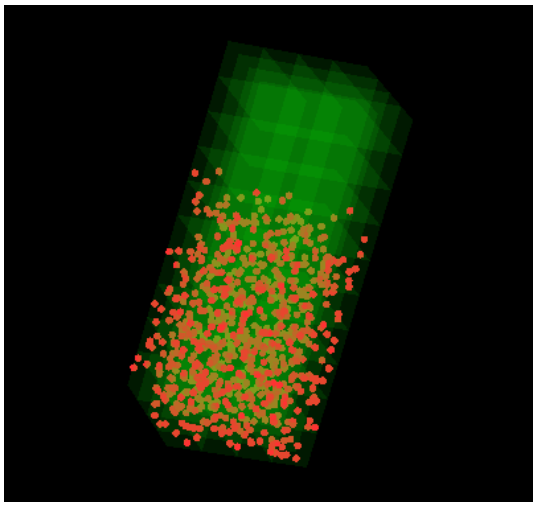

Figure 1: Model visualisation

region processes a single voxel, and use the terms voxel and model region interchangeably. (In reality, model regions typically process more than one voxel, and the HLA and Grid versions of BACGRID facilitate distribution by mapping 'non-local' voxel to voxel communication into HLA or Grid invocations, see below.) We give a high level description of a single iteration of the model. This illustrates the operation of both the diffusion module and the voxels (model regions) and how they interact to update the state of the model.

The state of the model at a given timestep t consists of the state of all the particles and voxels at that timestep. The state of each voxel e is given by the number of particles of each biomass type it contains $n_{b,e}$, and the concentrations of each substrate $c_{s,e}$ and signalling molecule $a_{q,e}$.[2] The state of each particle j is given by its biomass type b_j, its mass m_j, the number of up- u_j and down-regulated d_j cells it represents, its containing voxel e and its (notional) 3D position within that voxel.

Determining the state of the model at $t + 1$ involves determining, for each voxel, the change in substrate and signalling molecule concentrations due to diffusion $dc_{s,e}$, consumption $k_{s,e}$ (in the case of substrate) and production $z_{q,j}$ (in the case of signalling molecule), and for each particle, its change in mass over the timestep dm_j (and hence the change in the number of cells the particle represents dn_j), and the change in the number of up- du_j and down-regulated δd_j cells it contains and the particle's containing voxel.

The processing of the voxels at timestep t occurs in two phases. The first involves the execution of the voxels to calculate the consumption of substrate by particles, particle growth and the number of particles following division of the biomass. Processing of phase one within each voxel itself occurs in three steps. Firstly, the growth step increases the mass of each particle within each voxel given the concentration of substrate for this timestep in the voxel. (For $t = 0$, the concentrations and number of particles are taken as parameters of the simulation.) This also gives the total consumption of all

[2] See [3] for the definitions of symbols and equations.

substrates by all particles in this voxel for this timestep.

$$\boldsymbol{k}_e = Growth(\boldsymbol{c}_e)$$

where $Growth(\mathbf{c}_e)$ is given by equation (3.6) for each biomass type b in the voxel e. The second step computes the production of signalling molecule by each particle in the voxel.

$$\mathbf{z}_e = QSM(\boldsymbol{a}_e)$$

where $QSM(\boldsymbol{a}_e)$ is given by equation (3.13). The third step is particle division: each particle which reached the maximum allowable mass during the growth step is split into two particles, increasing the number of particles in the voxel.

$$\boldsymbol{n}_e = Division(Growth(\boldsymbol{c}_e))$$

Each voxel then sends the consumption of each type of substrate and the amount of QSM produced by its particles to the diffusion module. At the same time, each voxel sends its current particle counts to each of its neighbouring voxels.

The second phase of the timestep involves computing the changes in substrate and signalling molecule concentrations due to diffusion. The diffusion module uses the substrate consumption and signalling molecule production for this timestep to calculate the new substrate and signalling molecule concentrations for the next timestep. The diffusion module then sends each voxel the substrate and QSM concentrations for the next, $t + 1$, timestep.

In parallel with the execution of the diffusion module, each voxel also executes a displacement step, which uses the difference in pressure between the voxel and each of its adjacent voxels (which each voxel calculates using the number of particles in each of its neighbouring voxels) to determine movement of particles between voxels. The (possibly empty) list of displaced particles is then sent to each of the neighbouring voxels. A snapshot of the state of a particle for migration purposes consists of its biomass type, mass, the number of up-regulated cells, the voxel from which the particle is being migrated and the direction in which it is being migrated. Once each voxel has received a list of transfer particles from all it's neighbours the voxel updates its particle counts for the next timestep. The timestep is then incremented and the cycle repeats with the voxels using the newly calculated concentrations and particle counts.

In the non-distributed version of BACGRID, communication between modules is implemented using procedure calls. However, to facilitate implementation of the distributed versions, the diffusion module maintains a local copy of the substrate and QSM concentrations in each voxel.

5 HLA Distribution

In this section we describe the HLA version of BACGRID which allows the distribution of BACGRID modules across multiple processors in a single cluster or at remote sites.

The High Level Architecture (HLA), is a framework for simulation reuse and interoperability originally developed by the US Defence Modelling and Simulation Office [5]. Using HLA, a large-scale distributed simulation can be constructed by linking together a number of geographically distributed simulation components (or *federates*) into a single, larger simulation (or *federation*). The federates may be written in different languages and run on different machines. HLA (with minor revisions) has been

adopted as an IEEE standard (IEEE 1516) [2] and as such is likely to be increasingly widely adopted within the simulation community. HLA-compliance will therefore be an increasingly important feature of the next generation of simulators, allowing interoperation with other simulations, re-use of simulation components and the distribution of simulations across multiple computers to increase the overall performance of a global simulation.

The HLA consists of two parts: a set of rules specifying how federates can interoperate, and a runtime infrastructure (RTI) which provides core simulation services to an HLA federation. Each HLA federation has a Federation Object Model (FOM) which specifies how communication between federates is achieved. The FOM consists of a set of object classes and a set of interaction classes. Each *object class* defines a (possibly empty) set of named data called attributes. Instances of these object classes and their associated attribute values are created by the federates to define the persistent state of the simulation. Federates evolve the state of an object instance in simulation time by supplying new values for its attributes. An *interaction* is a set of named data, called parameters, which forms a logical unit within the federation, e.g., an event within the simulation model. The data comprising an interaction is sent as a unit by a federate to the other federates in the federation. Unlike objects, interactions have no continued existence after they have been received. Object and interaction classes are organised into (separate) inheritance hierarchies, in which each class inherits the attributes (for objects) or parameters (for interactions) of its superclasses. Each federate must typically translate from its internal notion of simulated entities to HLA objects and interactions as specified in the FOM. The structure of all FOMs is defined by the Object Model Template (OMT) which ensures federations can communicate with one another.

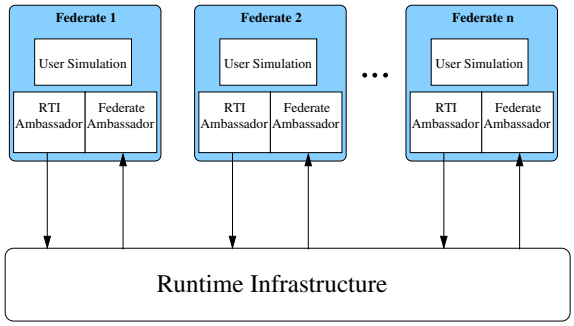

Figure 2: Architecture of federates in an HLA federation

The RTI is middleware that provides common services to the federates. All communication between the federates in a federation and between federations is accomplished via the RTI. Each federate contains an RTI Ambassador and a Federate Ambassador along with the user simulation code (see Figure 2). The RTI Ambassador handles all outgoing information passed from the user simulation to the RTI. Each call made by the RTI Ambassador typically results in a corresponding callback on other federates. For example, updating the value of an attribute of an instance of an object class defined in the FOM on one federate will result in a callback containing the new value on federates which subscribe to the attribute. It is the task of the Federate Ambassador to handle these callbacks and invoke appropriate code in the user simulation, e.g., update a the

value of a field or variable representing the attribute.[3] The FOM is passed to the RTI at the beginning of an execution and effectively parameterises the RTI for that federation.

The HLA provides services in six areas, namely Federation Management, Object Management, Declaration Management, Ownership Management, Data Distribution Management, and Time Management. In the remainder of this section, we illustrate the role of Object and Declaration Management, Data Distribution Management and Time Management in distributing a BACGRID simulation.[4]

In this section, we describe three aspects of the HLA version of BACGRID in more detail:

1. the types of federate and their roles in distributing the simulation;
2. the way the federates communicate with one another; and
3. how the model timestep relates to HLA time.

5.1 BACGRID Federates

In the HLA version of BACGRID, the modules described in section 3 are implemented as federates which communicate via HLA RTI calls. The federates act as wrappers for the corresponding BACGRID modules and handle communication with other federates running on different processors.

A BACGRID federation consists of two types of federate: a diffusion federate and one or more model region federates. The *diffusion federate* is a wrapper for the diffusion module, and as such is responsible for diffusion calculations throughout the entire computational domain.[5] Each *model region federate* wraps a model region (i.e., a MASON process responsible for simulating a contiguous collection of voxels), and handles communication with the diffusion federate and with adjoining model region federates (for pressure calculations and particle displacement). In the current prototype, a BACGRID federation consists of $m + 1$ federates: m model region federates and one diffusion federate.

5.1.1 Diffusion federate

The diffusion federate maintains a local record for each voxel in the system containing its position, substrate and signalling molecule concentrations and the model region to which the voxel belongs. During the simulation each model region federate sends changes in concentrations of substrates and signalling molecules to the diffusion federate. When the diffusion federate has received this information for all voxels in the system, it executes its diffusion algorithm until steady state is achieved. At this point the diffusion federate reports the new concentrations of substrate and signalling molecules back to each of the model region federates which in turn update the voxels.

5.1.2 Model region federate

The model region federates not only interact with the diffusion federate but also with other model regions federates. The voxels within each model region execute in two

[3] The RTI and Federate Ambassadors in HLA version of the BACGRID prototype utilise the DMSO RTI 1.3NGv6 Java bindings.

[4] We do not consider Federation Management in BACGRID, as this is similar to other HLA federations, and Ownership Management is not used in BACGRID.

[5] To date, we have not investigated the distribution of the diffusion module across multiple federates.

phases. The first phase grows and divides particles within the voxels, resulting in substrate consumption, signalling molecule production and particle division. The new substrate and signalling molecule concentrations are sent to the diffusion federate.

Particle division results in an increase in the number of particles and hence the pressure in each voxel. Once the new pressures have been calculated, each voxel uses the difference between its own pressure and that of each of its neighbours to calculate the number of particles which must be transferred to equalise the pressure. For most voxels within a model region, its neighbouring voxels are managed by the same the same model region federate. However, for those voxels which lie on the boundaries of a model region, some of the particle counts needed to calculate the pressure differences are associated with voxels managed by neighbouring model regions. Each model region federate therefore maintains *proxy voxels* for those voxels in neighbouring model regions which are adjacent to its boundary voxels. Figure 3 shows two adjacent model regions and their overlapping proxy voxels. Voxel b in model region 2 has a corresponding proxy voxel (proxy b) in model region 1. The arrow indicates the direction of communication between the voxel and its proxy. Note that communication is one way—model region 1 does not update proxy b. The state of proxy b (its particle count) is updated by model region 2 once voxel b has finished its growth and division step. As the x and y boundaries of the computational domain are periodic, the model regions at each domain boundary maintain proxies for the corresponding voxels at the opposite extreme of the computational domain. The z axis boundaries are treated as a special case, with a particle count of infinity being returned for voxels 'below' $z = 0$, and a particle count of zero for voxels 'above' $z = L_Z$.

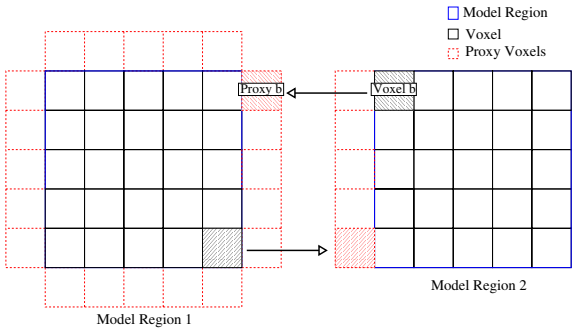

Figure 3: Two model regions and the proxy voxel overlap

At the end of the first phase of execution, the model regions send particle count updates for their boundary voxels to each of their neighbouring model region federates. Each message contains a list of pairs each containing the voxel id and the new particle count, and the information is used to update the receiving model region's proxy voxels. Once the relative pressures have been determined, each voxel computes the number of particles to transfer in each direction. A list of particles to transfer is then passed to the model region which determines if the receiving voxel is local. If the transfer is between two voxels on the same model region, the particles and their state are transferred through method calls. However, if the receiving voxel is remote, the model region packages up the particle ready for transmission. Once all the particles to be transferred in a given direction have been assembled, the model region sends the particles to the adjacent model region, where they are unpacked and new particles

created in the appropriate voxels.

5.2 Communication Between Federates

Information is exchanged in 3 different ways within a BACGRID federation:

1. Model Region Federate → Diffusion Federate: consumption of substrate(s) and production of QSM(s);
2. Diffusion Federate → Model Region Federate: substrate and QSM concentrations; and
3. Model Region Federate → Model Region Federate: particle counts and particle transfer.

In the remainder of this section we outline how HLA services are used to implement communication between the federates in a BACGRID federation.

5.2.1 Object and Declaration Management

To reduce the number of RTI calls (and ultimately the number of Grid service invocations in HLA_GRID), we chose to implement the communication between federates using interactions rather than updates of object attribute values. While attribute values are arguably a more natural realisation of the model state, interactions allow greater flexibility in communication. For example, if voxels are modelled as objects, the number of attribute updates required for communication between the diffusion and model region federates is linear in the number of voxels, and quadratic in the number of voxels on one side of a model region for communication between model region federates. In contrast, the number of interactions required for communication between the diffusion and model region federates is linear in the number of model regions and constant for communication between model regions. Interactions also make it easier to package data up into larger messages.

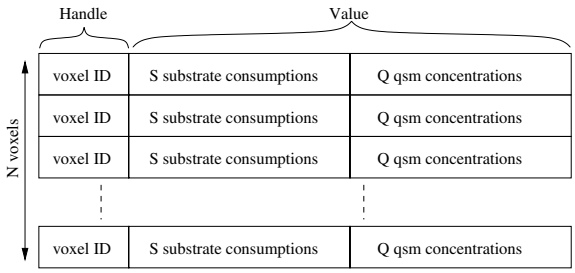

Figure 4: Structure of a `modelToDiffuser` interaction

Four interaction classes were defined, all subclasses of an (abstract) `BacGridInteraction` class:

- `modelToDiffuser`;
- `diffuserToModel`;
- `particleCount`; and

- `particleTransfer`.

`modelToDiffuser` and `diffuserToModel` interactions are used to transfer substrate and signalling molecule concentrations from the model region federates to the diffusion federate and vice versa. `particleCount` and `particleTransfer` interactions are used to transfer particle counts between the boundary voxels of a model region federate and their proxies managed by the neighbouring model region federates, and to transfer particle state information between model region federates during particle displacement. In each case, a single interaction is used to transfer the relevant information for all voxels in a model region (or all the boundary voxels on one face of a model region in the case of `particleCount` and `particleTransfer` interactions). For example, figure 4 illustrates the structure of a single `modelToDiffuser` interaction. The content of the interaction is a `handleValuePairSet`, with voxel IDs as handles and a list of substrate and signalling molecule concentrations as the value. If each concentration is represented as a double (8 bytes) and a voxel's 3D position is used as its ID (3 ints, 12 bytes), the total size of a `modelToDiffuser` interaction is:

$$N(12 + 8(S + Q))$$

bytes, where S is the number of substrates, Q is the number of signalling molecule types and N is the number of voxels managed by the model region federate.

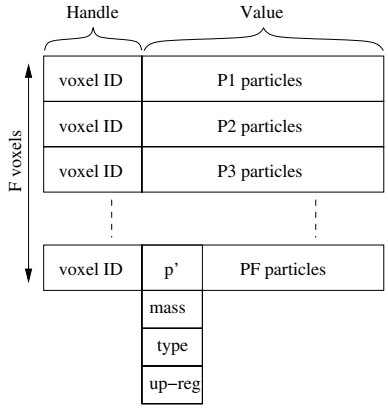

Figure 5: Structure of a `particleTransfer` interaction

Similarly, each model region sends a particle transfer interaction to each of its adjacent model regions at every timestep. The structure of `particleTransfer` interactions (see Figure 5) is similar to `modelToDiffuser` interactions, with the voxel ID as the handle and a list of particles and their state as the value. The total size of a single particle transfer interaction depends on the number of particles transferred at a given timestep. For example, if we assume each of the n^2 boundary voxels transfer p particles each timestep, and the voxel ID is represented as 3 ints (12 bytes) as before, the biomass type by a short (2 bytes), the mass of the particle by a double (8 bytes), and the number of up-regulated cells by an int (4 bytes) then the the size of single `particleTransfer` interaction is:

$$n^2(12 + P(8 + 2 + 4))$$

In the HLA, a federate declares its interest in objects and interactions at the beginning of a simulation by *publishing* any attributes it may update or interactions it may send during the simulation and *subscribing* to attributes which it would like to receive updates for and interactions it would like to receive. In the BACGRID federation the publication and subscription of each federate is straightforward. The diffusion federate publishes `diffuserToModel` interactions, and subscribes to `modelToDiffuser` interactions. A model region federate publishes `modelToDiffuser`, `particleCount` and `particleTransfer` interactions, and subscribes to `diffuserToModel`, `particleCount` and `particleTransfer` interactions.

5.2.2 Data Distribution Management

With the publications and subscriptions outlined in the previous section, each model region would receive substrate and QSM concentrations for all model regions in the system.[6] To minimise network traffic, BACGRID uses the HLA Data Distribution Management (DDM) services to constrain the subscription of each federate. Each model region therefore only receives interactions which contain information about its state.

As noted in section 5.2.1, information is exchanged in three ways within a BACGRID federation:

1. Model Region Federate → Diffusion Federate: consumption of substrate(s) and production of QSM(s);
2. Diffusion Federate → Model Region Federate: substrate and QSM concentrations; and
3. Model Region Federate → Model Region Federate: particle counts and particle transfer.

In the case of transfers from model region federates to the diffusion federate, no DDM is necessary, as the diffusion federate receives this information from all model regions.[7]

The second type of information exchange is from the diffusion federate to the model region federates. In this case DDM is required to ensure each model region only receives new concentrations for the voxels it manages. To do this we define a 3D *routing space* of model regions, and have each model region specify a single point in the routing space, namely its own position, when subscribing to `diffuserToModel` interactions. The position of each model region is defined in relation to the other model regions in the domain. When the diffusion federate sends the interactions containing the concentrations of substrates and QSMs, it specifies the point in the routing space corresponding to the position of the model region which the updates are intended for.

Particle counts are transferred from model region federate to model region federate, with each model region federate sending the particle counts of its boundary voxels to the neighbouring model region federates. To enable this, each model region federate defines six DDM regions for particle counts, one for each face of the model region it manages. A neighbouring model region federate subscribes to the DDM region associated with the adjoining face. For example, Figure 6 shows a plan view of a model

[6] Since in HLA there is no way for a federate to know how many other federates there are in the federation or which objects and interactions they are currently subscribed to, there is no way to specify a particular federate as the recipient of an interaction or attribute update.

[7] If the diffusion algorithm were distributed across multiple federates, this would no longer be the case.

region federate and the DDM subscription regions of two of its neighbouring model region federates. In the figure, the federate model region 1 has two DDM regions (2 and 3) which are subscribed to by the federates model region 2 and 3. In addition, the federate model region 1 also subscribes to corresponding DDM regions in the neighbouring model regions (not shown). The same DDM regions can be used to manage distribution of `particleTransfer` interactions between model region federates.

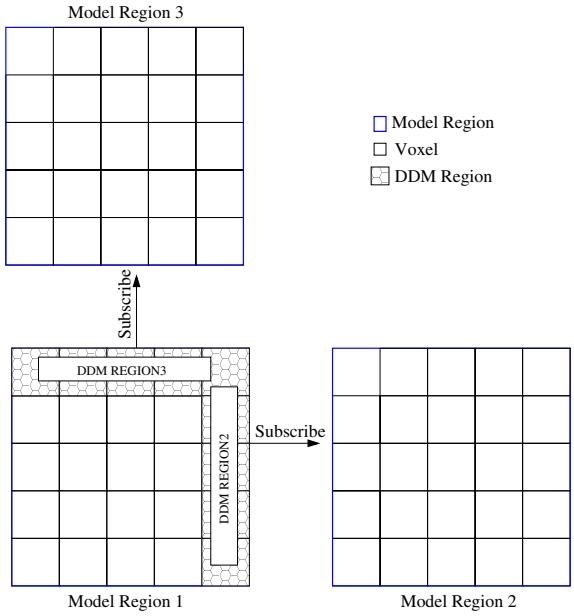

Figure 6: Three model regions and the the DDM regions used for interactions

5.2.3 Time Management

As described in section 4 (and in detail in [3]), the model timestep consists of two main phases each comprising a number of steps. In this section we sketch a single iteration of the model timestep in BACGRID, indicating which computations execute in parallel, the points at which information is exchanged between the diffusion and model region federate(s) and how federates synchronise using HLA time management services.

Each model timestep begins with the model region federates executing their voxels. Each voxel in turn executes its particles' growth and division step. All model region federates execute in parallel, however voxels and particles are executed sequentially within each model region. Once the growth and division step is complete, each model region has new values for substrate and signalling molecule concentrations and particle counts. Each model region federate then sends an interaction containing the substrate consumption and and signalling molecule production for each of its voxels at this timestep, which is received by the diffusion federate. It also sends the particle counts for its boundary voxels to the DDM regions corresponding to its neighbouring model region federates.

At this point phase two of timestep begins. Once the diffusion federate receives `modelToDiffuser` interactions from all model region federates, it executes the dif-

fusion algorithm until steady state is achieved. The diffusion federate then sends an interaction containing the new substrate and signalling molecule concentrations for each model region federate to the appropriate DDM region. In parallel, each model region federate calculates the number of particles to transfer between local and remote voxels using the particle counts computed during phase one. It then determines which particles are to be transferred to voxels managed by neighbouring model region federates and sends an interaction to the DDM region corresponding to each neighbouring model region federate containing the state(s) of the transferred particle(s). The diffusion federate (diffusion algorithm) and the model region federates (pressure calculations and particle transfer) execute concurrently in phase two.

Each model timestep corresponds to two HLA timesteps. Each exchange of information (via an interaction) has an associated timestamp. All interactions generated in phase one have timestamp $t + 1$. Once a model region federate has consumption of substrates and production of QSMs to the diffusion federate and particle counts to its neighbouring model region federates, it makes an HLA time advance request to time $t + 1$. The diffusion federate also makes a time advance request to $t + 1$ at the beginning of the model timestep.[8] All interactions generated in phase two have a timestamp of $t + 2$. Once the diffusion federate has sent an interaction containing the updated substrate and signalling molecule concentrations for the next model timestep to each model region federate, it makes an HLA time advance request to time $t + 2$. Similarly each model region federate makes a time advance request to $t + 2$ once it has sent an interaction to each neighbouring model region federate containing the particles to be transferred at this model timestep. When time advances to $t + 2$ each model region federate updates their boundary voxels with any particles transferred from neighbouring model regions. This completes processing at this model timestep, and all federates then advance to the next model timestep.

5.3 HLA Implementation

The HLA distribution is based on the DMSO RTI 1.3NGv6 Java bindings from the DMSO RTI reference distribution. Java classes were defined to implement the `BacGridInteraction`, `modelToDiffuser`, `diffuserToModel`, `particleCount` and `particleTransfer` interactions. Other classes, e.g., classes representing logical times, were provided by the DMSO RTI 1.3NGv6 Java bindings, as were the implementations of the RTI and Federate Ambassadors.

6 Grid Distribution

The Grid version of BACGRID extends the HLA version to allow the inter-operation of federates using Grid services. The use of Grid services has a number of advantages compared to distribution using the standard HLA protocols. In order to run a distributed simulation using HLA, special arrangements have to be made beforehand to ensure the availability of the required hardware and software. Such arrangements typically imply some form of centralised control, as the inter-organisational sharing of resources involved impacts issues such as security and resource allocation policies [12]. For example, significant firewall configuration is required if the standard HLA protocols are used. In contrast, Grid services support secure, scalable inter-operation of simulation components, simplifying coordination and management of simulations.

[8] The diffusion federate remains blocked until the model region federates request time advance to $t + 1$.

The BACGRID Grid distribution is based on HLA_GRID [12], which allows HLA-compliant simulators to be instantiated and linked using Grid services. HLA_GRID employs a Federate-Proxy-RTI architecture, in which a proxy acts on behalf of the federate in interacting with the RTI. Participants (client federates) in a simulation run their federate codes at their local sites, and proxies and the RTI is executed at remote Grid resources. RTI services are exposed as Grid services, and federate codes and their respective proxies communicate with each other through Grid services and a Grid-enabled HLA library, which provides the standard HLA API to the federate codes, and translates RTI calls into Grid service invocations. HLA_GRID includes additional Grid services to support the creation of the RTI, discovery of federations, etc.

HLA_GRID aims to improve the interoperability and composability of HLA-compliant simulation components using the facilities of the Grid [9]. In contrast to the standard HLA which requires that each federate be linked against the RTI libraries, clients need only the HLA_GRID library (which contains no RTI code) for a federate to be able to inter-operate with other federates. Users can therefore run simulations without having to deploy or manage the RTI. HLA_GRID itself is implemented in Java, allowing integration with simulators on a wide range of platforms. In addition, an entire simulation can be provided as a Grid service, which can be discovered and used via HLA_GRID, allowing hierarchical federations to be constructed using HLA_GRID proxies in a way similar to [].

At the current state of development, HLA_GRID allows users at different sites to cooperate in the development of a simulation which combines simulators which they have developed locally. It also allows users to combine their local simulator with a pre-existing simulation provided as a Grid service. As yet it does not allow simulation developers to offer simulation components as Grid services. In the longer term, each federate could manifest itself as a Grid service for use by the simulator. Such 'simulation Grid services' would allow the automatic composition of simulation components obtained by web service registries. However automated federate discovery and configuration requires the definition of appropriate ontologies and languages to represent federate metadata to allow the orchestration of simulations by matching user requirements with appropriate federates, and is outside the scope of the current project. See [9] for more details.

6.1 Architecture of an HLA_GRID Federate

In HLA_GRID, each federate consists of two parts: the 'federate proper' which executes on the client side, and the proxy which executes remotely. The client side contains the user-supplied federate code, the Client RTI Ambassador and the Client Federate Ambassador Service.[9] The client side components usually run on a local machine (from the simulation model's point of view). The proxy consists of the Proxy RTI Ambassador Service and the Proxy Federate Ambassador. These act on behalf of the client and interact with other proxies via the RTI, usually over a LAN at a remote site.

The Client RTI Ambassador provides a standard HLA API to the user federate code, and allows communication with the proxy via the Grid. RTI calls by the federate code result in remote Proxy RTI Ambassador Service invocations with method parameters embedded inside the invocation. The proxy is responsible for translating Proxy RTI

[9]The terms used to name the various parts of the HLA_GRID framework have evolved over time. In what follows, we use the terminology adopted in [9].

Ambassador Service invocations into normal federate initiated RTI calls, and embedding Federate Ambassador callbacks from the RTI into Client Federate Ambassador Service invocations. Both the Proxy RTI Ambassador Service and the Client Federate Ambassador Service are implemented as Grid services using the Globus toolkit. All the methods of the `RTI::RTIambassador` class and the callbacks provided by the `RTI::FederateAmbassador` class from the DMSO RTI 1.3NGv6 Java bindings are exposed to Globus. While both parts of the federate are coupled together (by their respective Grid services) to form a single 'federate executive', the proxy decouples the client and the RTI, with the simulation logic and state being maintained at the client side. The proxy only provides mechanisms for simulation management, such as federation management and time management.

In addition to the Grid services which enable communication between the client's federate code and the remote RTI, HLA_GRID provides Grid services for creating the RTI and discovering the federation: persistent RTI service factories are used to create instances of RTI services; and a persistent indexing service maintains the mapping between federations and handles of corresponding RTI services instances. Details of how these services are created, managed and coordinated can be found in [14].

When the simulation starts up, the federate code invokes the persistent RTI service factory to create an RTI service instance. The newly created RTI instance and the federation name are registered with the index service so that other federates in the same federation will be able to look up the correct RTI instance. The Proxy RTI Ambassador Services for each federate are then started. The identity of each federate's Proxy RTI Ambassador Service is passed to the corresponding Client RTI Ambassador. The Client Federate Ambassador Service for each federate is then initialised and registered with the corresponding Proxy Federate Ambassador.

Thereafter, simulation proceeds as in the HLA version of BACGRID. Simulation events and state changes generated by the user-supplied federate code result in RTI function calls to the Client RTI Ambassador, which translates them into Grid service invocations to access the remote Proxy RTI Ambassador Service on the proxy side. The Proxy RTI Ambassador Service interacts with the real RTI by executing the real RTI calls on behalf of the client. Return values are sent back as the return value of the Grid service invocation. Callbacks from the RTI to a federate are translated into invocations of the Client Federate Ambassador Service by the Proxy Federate Ambassador. The Client Federate Ambassador Service converts these into 'real' RTI callbacks to the federate.

Previous work [13] suggests that an RTI call in HLA_GRID incurs an overhead of about a factor of 5 compared to HLA. Where possible we have therefore designed the simulator to minimise the number of RTI and hence Grid service invocations (see section 5.2.1), while at the same time trying to maintain a straightforward mapping from biological to simulation concepts.

6.2 HLA_GRID Implementation

The original version of HLA_GRID was implemented at Nanyang Technological University and was based on Globus Toolkit version 3 (GT3)[10] (see Figure 7). It included standard HLA/RTI APIs to support Federation, Object, Declaration, Ownership and Time Management. For a more detailed description of the GT3 version of HLA_GRID, see [11, 12]

[10] See `http://www.globus.org/`

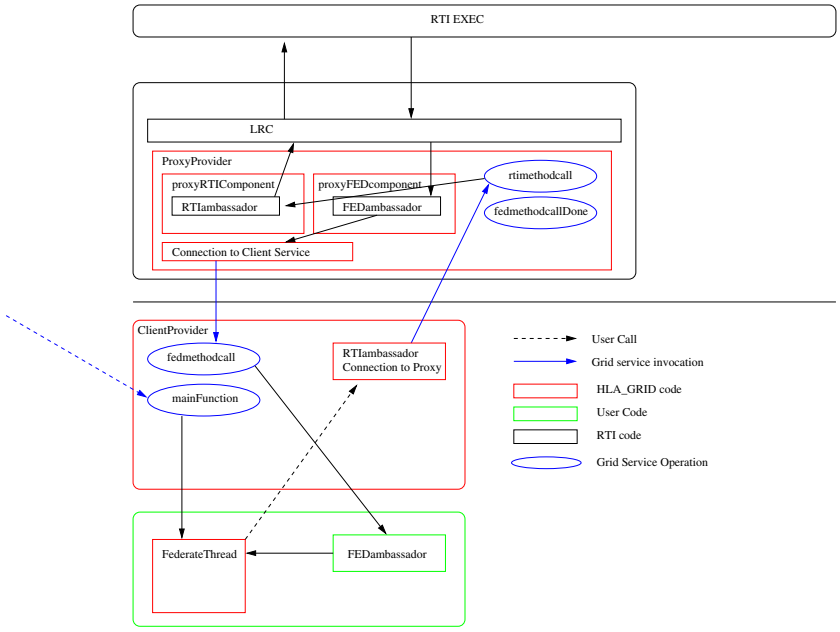

Figure 7: Architecture of HLA_GRID (GT3 version)

To allow HLA_GRID (and hence BACGRID) to be used on the current generation of Grid infrastructure, it was necessary to reimplement HLA_GRID for the current version of the Globus Toolkit (version 4). At the same time, we took the opportunity to refactor the design of HLA_GRID (see Figure 8). For example, the GT3 version of HLA_GRID was implemented as a single grid service, with the particular RTI service being indicated by a parameter to the Grid service invocation. In the GT4 version of HLA_GRID, each RTI service is exposed as a separate Grid service, and the arguments to the RTI calls are mapped to SOAP complex types by the client RTI Ambassador. For example, the RTI type `ParameterHandleValuePairSet` is mapped to a corresponding SOAP complex type. A full list of SOAP types can be found in table 1.

Type	Constituting types
`ArrayOfByte`	`byte[]`
`HandleSet`	`int[]`
`HandleValuePair`	`int, ArrayOfByte`
`EventRetractionHandle`	`int, int`
`HandleValuePairSet`	`HandleValuePair[]`
`Time`	`ArrayOfByte`

Table 1: SOAP complex types used in the HLA version of BACGRID

As noted in [7] there are particular problems in implementing RTI DDM services as Web or Grid services. As a result, the GT3 version of HLA_GRID (described in [12]) does not implement the Data Distribution Management services of the RTI. The benefits of DDM for a BACGRID federation have yet to be quantified, and is not clear if the additional effort required to implement DDM in the GT4 version of HLA_GRID

is warranted. We have therefore decided not to implement DDM services in the initial GT4 version of HLA_GRID.

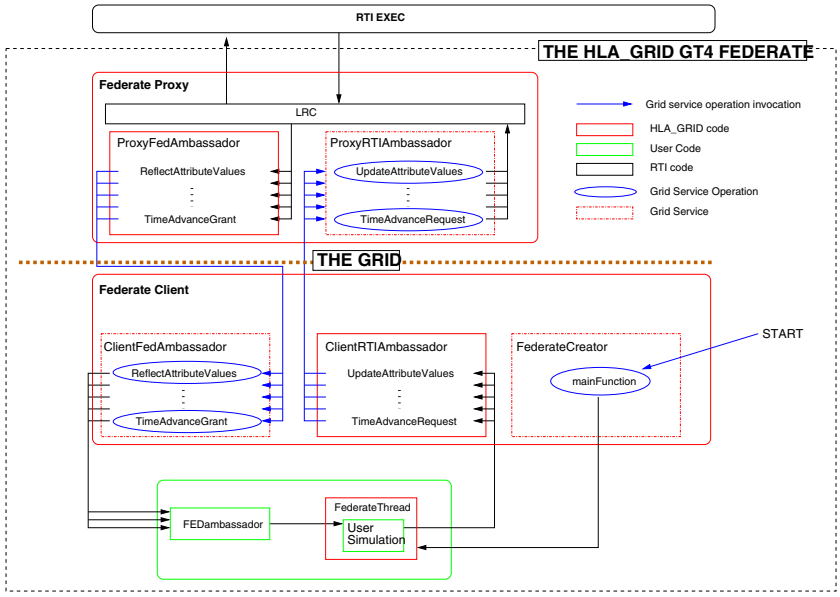

Figure 8: Architecture of HLA_GRID (GT4 version)

The GT3 version of HLA_GRID used in the experiments reported in [13] included additional support for remote exception handling. The Proxy RTI Ambassador Service was extended to allow exceptions generated by RTI calls to be filtered at the proxy side. After communication with the Proxy RTI Ambassador Service is established, the Client RTI Ambassador registers 'uninteresting' exceptions with Proxy RTI Ambassador Service: only unregistered exceptions are returned to the client as Apache Axis faults. Registered RTI call exceptions are handled (discarded) remotely at the proxy side. These extensions were necessitated by the particular requirements of the HLA_REPAST agent toolkit used in these experiments, and support for remote exception filtering has not been reimplemented in the GT4 version of HLA_GRID developed at Nottingham.

References

[1] H. M. Byrne, J. R. King, D. L. S. McElwain, and L. Preziosi. A two-phase model of solid tumour growth. *Applied Mathematics Letters*, 16(4):567–573, 2003.

[2] IEEE Standard for modeling and simulation (M&S) High Level Architecture (HLA) — Framework and rules. IEEE, 2000. (IEEE Standard No.: 1516-2000).

[3] John King, Michael Lees, and Brian Logan. Agent-based and continuum modelling of populations of cells. Technical report, University of Nottingham, December 2006.

[4] Jan-Ulrich Kreft, Ginger Booth, and Julian W. T. Wimpenny. BacSim, a simulator for individual-based modelling of bacterial colony growth. *Microbiology*, 144:3275–3287, 1998.

[5] Fredderick Kuhl, Richard. Weatherly, and Judith Dahmann. *Creating Computer Simulation Systems: An Introduction to the High Level Architecture*. Prentice Hall, 1999.

[6] Sean Luke, Claudio Cioffi-Revilla, Liviu Panait, Keith Sullivan, and Gabriel Balan. MASON: A multiagent simulation environment. *Simulation*, 81(7):517–527, 2005.

[7] Katherine L. Morse, David L. Drake, and Ryan P. Z. Brunton. Web enabling an RTI — an XMSF profile. In *Proceedings of the 2003 European Simulation Interoperability Workshop*. European Office of Aerospace R&D, Simulation Interoperability Standards Organisation and Society for Computer Simulation International, June 2003. Paper number 03E–SIW–063.

[8] Cristian Picioreanu, Jan-Ulrich Kreft, and Mark C. M. van Loosdrecht. Particle-based multidimensional multispecies biofilm model. *Applied and Environmental Microbiology*, 70(5):3024–3040, May 2004.

[9] Georgios Theodoropoulos, Yi Zhang, Dan Chen, Rob Minson, Stephen John Turner, Wentong Cai, Yong Xie, and Brian Logan. Large scale distributed simulation on the grid. In *Sixth IEEE International Symposium on Cluster Computing and the Grid Workshops (CCGRIDW'06)*, page 63, Singapore, May 2006. IEEE Computer Society.

[10] J. P. Ward, J. R. King, A. J. Koerber, J. M. Croft, R. E. Sockett, and P. Williams. Early development and quorum sensing in bacterial biofilms. *Journal of Mathematical Biology*, (47):23–55, 2003.

[11] Y. Xie, Y. M. Teo, W. Cai, and S. Turner. A distributed simulation backbone for executing HLA-based simulation over the Internet. In *Workshop on Grid Computing and Applications, Proceedings of the Second International Conference on Scientific and Engineering Computation*, pages 96–103, Monterey, CA, USA, June 2004.

[12] Y. Xie, Y. M. Teo, W. Cai, and S. Turner. Service provisioning for HLA-based distributed simulation on the Grid. In *Proceedings of the Nineteenth ACM/IEEE/SCS Workshop on Principles of Advanced and Distributed Simulation (PADS 2005)*, pages 282–291, Monterey, CA, USA, June 2005.

[13] Yi Zhang, Georgios Theodoropoulos, Rob Minson, Stephen Turner, Wentong Cai, Yong Xie, and Brian Logan. Grid-aware large scale distributed simulation of agent-based systems. In *Proceedings of the 2005 European Simulation Interoperability Workshop*, Toulouse, June 2005. Simulation Interoperability Standards Organisation, IEEE/ITCMS. 05E-SIW-047.

[14] Wenbo Zong, Yong Wang, Wentong Cai, and Stephen J. Turner. Grid services and service discovery for HLA-based distributed simulation. In *Proceedings of the Eighth IEEE International Symposium on Distributed Simulation and Real-Time Applications (DS-RT'04)*, pages 116–124, Washington, DC, USA, 2004. IEEE Computer Society.