Defence Research and Development Canada
Recherche et développement pour la défense Canada

DEFENCE R&D DÉFENSE

IPME and external clients:

Enhancing simulation performance by delegating workload to external clients; explaining and simplifying the process

Phil ter Haar

Brad Cain

Defence R&D Canada
Technical Memorandum
DRDC Toronto TM 2007-033
December 2007

Canada

IPME and external clients:

Enhancing simulation performance by delegating workload to external clients; explaining and simplifying the process

Phil ter Haar

Brad Cain

Defence R&D Canada – Toronto

Technical Memorandum
DRDC Toronto TM 2007-033
December 2007

Principal Author

Original signed by Phil ter Haar

Phil ter Haar

Research Technologist

Approved by

Original signed by Keith Niall

Keith Niall

Acting Head, Human Systems Integration Section

Approved for release by

Original signed by K. C. Wulterkens

K. C. Wulterkens

for Chair, Document Review and Library Committee

Abstract

The speed of simulation execution as performed by the Integrated Performance Modelling Environment (IPME) degrades significantly when computationally demanding functions, such as iterative mathematical calculations, are included in a model. Much of this degradation can be mitigated by transferring those functions that require significant computational resources (cpu processing) to an external client using the external client architecture that accompanies IPME. This client communicates with IPME using TCP/IP network protocol, exchanging values of common variables over the network. Using an external client allows more processing power to be dedicated to computationally expensive tasks and are generally more robust to increases in computational demand. A computationally demanding sample client is used to show the execution performance differences when the procedure resides within the IPME task network and when it is offloaded to an external client. This report also outlines how to use the sample client source code to build a client program, extending the developer's approach for client program development to create a more flexible interface. This will provide another option for client developers who may prefer a more intuitive template for developing their customized client than the sample client provided with IPME.

Résumé

Les performances de l'environnement intégré de modélisation des performances (EIMP) se dégradent considérablement lorsque des fonctions exigeant beaucoup de puissance de calcul sont intégrées dans un modèle. Cette dégradation peut être atténuée considérablement en transférant la fonction exigeant beaucoup de puissance à un client externe qui utilise l'architecture de client externe connexe à l'EIMP. Ce client communique avec l'EIMP au moyen du protocole réseau TCP/IP, échangeant ainsi des valeurs de variables partagées par le réseau. L'utilisation d'un client externe permet de réserver plus de puissance de traitement à la tâche exigeant beaucoup de puissance de calcul. Cela permet aussi de mieux répondre aux augmentations de la demande visant ces tâches. Un exemple de client exigeant beaucoup de puissance est utilisé pour faire la démonstration de la différence sur le plan des performances à l'exécution entre une procédure résidant dans le réseau d'exécution de tâche de l'environnement EIMP et la même procédure transférée à un client externe. Ce rapport décrit aussi comment utiliser le code source de l'exemple de client pour produire un programme client, ce qui étend la portée de la démarche du développeur en matière de développement de programmes clients afin de pouvoir créer une interface plus souple.

This page intentionally left blank.

Executive summary

IPME and external clients: Enhancing simulation performance by delegating workload to external clients; explaining and simplifying the process

[Phil ter Haar; Brad Cain]; DRDC Toronto TM 2007-033; Defence R&D Canada – Toronto; December 2007.

Introduction: The Integrated Performance Modelling Environment (IPME) is a network simulation software package for building models that simulate human and system performance (anonymous, 2004). Though well equipped with tools and utilities that allow for great fidelity in discrete event modelling, version 3 of IPME is limited in its capability to perform large amounts of computations at a speed fast enough to suit the requirements of some applications. Fortunately, IPME is designed to interface with external programs at runtime, allowing custom routines to be added in a modular manner, independent of the IPME model. It is the purpose of this investigation to compare the performance of a computationally intensive model across two architectural configurations; 1) with the demanding procedure embedded within IPME (*standalone* condition) versus 2) hosting the same demanding procedure outside IPME using an external client (*with-client* condition). By running trials of increasing computational workload we expect to see a trend of superior performance in the with-client condition as the workload intensity increases. Any increased effeciencies related to the with-client condition can be attributed to the superior processing abilities of external pre-compiled programs relative to IPME's internal code execution with is interpreted at run-time.

The steps to create a client program are discussed, based on a sample client program that is packaged with the IPME software. The typical approach to building a customized client program is to start from this code but changes to clients typically require recompiling the client program. In some cases, it is only the exchanged variables that differ among applications and the structure of the client code remains unchanged. A programming approach called the 'Client Function Module' (CFM) has been developed to extend the versatility of external IPME clients by using parameter files, eliminating the need to recompile. This technique does not provide increased performance or efficiency rather it provides a modular framework that allows a programmer to remain organized and modify only the parts of the code that are necessary. The client used in this report demonstrates this approach.

Results: Relocating the computationally demanding procedure to an external client resulted in a substantial enhancement of performance in the IPME simulation, showing from one to two orders of magnitude improvement in execution speed. Not only was the overall performance better using the 'with-client' configuration, but it was also found that the external client was less sensitive to increases in computational demand than when the code was embedded in IPME..

Significance: IPME modellers can improve the performance of their simulations by moving computationally internal procedures to external clients to reduce overall execution time, achieving real time or faster than real time execution. This will allow more complex models of human performance to be executed with reduced time demands, extending the capability to

predict human-system performance in greater detail or of greater scope. As systems become more complex, testing by simulation will become increasingly important to ensure that the desired performance is achieved. Clients provide a mechanism to support this activity.

Future plans: IPME version 4 is being developed and should be more computationally powerful and robust to workload increases. While the role of external clients as performance enhancers decrease, they will continue to lend support in a capacity of providing modular functionality to IPME models. The CFM approach can be improved in terms of compilation. The ipme_interface.c is currently compiled along with the CFM code, but it need not be. Future development could see it compiled into a library that is simply linked to the customized CFM module, serving its IPME interfacing needs.

Sommaire

Amélioration des performances par délestage de la charge de travail à des clients externes : explication et simplification du processus

[Phil ter Haar; Brad Cain]; DRDC Toronto TM 2007-033; R & D pour la défense Canada – Toronto; Décembre 2007.

Introduction ou contexte : L'environnement intégré de modélisation des performances (EIMP) est un ensemble de logiciels de simulation réseau utilisé afin de développer des modèles qui simulent les activités des personnes et des systèmes (anonyme, 2004). Bien qu'elle soit bien pourvue en outils et en utilitaires permettant la réalisation de modélisations d'événements discrets d'une grande fidélité, la version 3 de l'EIMP est limitée sur le plan de l'exécution de grandes quantités de calculs à une vitesse suffisante pour répondre aux besoins de certaines applications. Heureusement, l'EIMP est conçu afin de pouvoir être interfacé avec des programmes externes pendant son exécution, ce qui permet l'ajout de sous-programmes sous forme modulaire, indépendamment du modèle EIMP. La présente étude vise à comparer les performances d'un modèle exigeant beaucoup de calcul et intégrant la procédure lourde dans l'EIMP (état autonome) avec le même modèle dont la procédure lourde serait intégrée à un client externe. En effectuant des essais avec des charges de travail de calcul croissantes, nous nous attendons à relever une tendance vers l'obtention de performances supérieures au moyen du modèle avec client, performances qui augmenteront avec la croissance de la charge de travail.

Les étapes de création d'un programme client qui sont décrites portent sur un exemple de programme client qui est distribué avec l'EIMP. La démarche habituelle de construction d'un programme client consiste à commencer par le développement de ce code, mais les changements que l'on souhaite apporter au client exigent habituellement de recompiler le client. Dans certains cas, ce sont seulement les variables échangées qui diffèrent d'un programme à l'autre et la structure du code demeure inchangée. Une procédure de programmation, fondée sur le « module de fonction client » (CFM en anglais), a été élaborée afin d'accroître la versatilité des clients EIMP externes en ayant recours à des fichiers de paramètres, ce qui élimine le besoin de recompiler. Cette technique n'améliore pas les performances ou l'efficience, mais elle constitue un cadre permettant à un programmeur de rester ordonné et de modifier seulement les parties du code requises. Le client utilisé dans le présent rapport relève de cette approche.

Résultats : La relocalisation d'une procédure exigeant beaucoup de puissance de calcul à un client externe a permis d'obtenir une amélioration considérable des performances de la simulation en environnement EIMP. On a ainsi obtenu une accélération de la vitesse d'exécution de un à deux ordres de grandeur. Non seulement la performance d'ensemble a-t-elle été améliorée au moyen de la version « avec client », mais on a aussi remarqué que le client était moins sensible aux augmentations de la demande de calcul que lorsque le code était intégré à l'environnement EIMP.

Importance : Les modélisateurs qui utilisent EIMP peuvent améliorer la performance de leurs simulations en déplaçant les procédures exigeant du calcul à des clients externes afin de réduire le

temps d'exécution d'ensemble, atteignant ainsi des vitesses d'exécution égalant le temps réel ou plus rapides que le temps réel. Cela permettra de réaliser des modèles de performances humaines plus complexes, mais exigeant moins de temps, ce qui accroîtra la capacité de prévoir les performances des ensembles humain-système plus en détail, avec une plus grande portée. À mesure que les systèmes se complexifient, il devient de plus en plus important d'effectuer des essais par simulation afin de vérifier que l'on obtient les performances voulues. Le recours aux clients aide à la réalisation de cette activité.

Perspectives : La version 4 de l'EIMP est en cours de développement et elle devrait être plus puissante sur le plan de la capacité de calcul et plus résistante à l'augmentation de la charge de travail. Bien que le rôle des clients externes à titre de dispositif améliorant les performances ira en diminuant, ils continueront d'aider à assurer la modularité des modèles de l'EIMP. La démarche qui fait appel au module de fonction client peut être améliorée sur le plan de la compilation. L'interface EIMP est actuellement compilée avec le code du module de fonction client, mais cela n'est pas essentiel. Le développement futur pourrait la voir compilée dans une bibliothèque qui est simplement liée au module de fonction client personnalisé, répondant ainsi au besoin qu'il soit interfacé avec l'EIMP.

Table of contents

Abstract …….. i
Résumé ……... i
Executive summary iii
Sommaire v
Table of contents vii
List of figures ix
List of tables x
1 IPME and the external client 1
2 IPME client socket communication 2
2.1 IPME sample client 2
2.1.1 Overview 2
2.1.2 The code 3
2.1.3 Shared variables 3
2.1.4 Incoming variables 3
2.1.5 Outgoing variables 4
2.2 Timing parameters 4
2.2.1 Configuring client event timing 4
2.2.1.1 Client timing variables 5
2.2.1.2 Simple example 5
2.2.1.3 Complex example 1 6
2.2.1.4 Complex example 2 7
2.2.1.5 Complex example 3 7
2.2.1.6 Conclusion 8
2.3 A generic external client interface 8
2.3.1 The issue 8
2.3.2 A solution 9
2.3.2.1 CFM function descriptions 9
3 Performance enhancements using clients 12
3.1 Overview 12
3.2 Setting up the ‘standalone’ mode 12
3.3 Setting up the ‘with-client’ mode 13
3.4 Experimental design 16
3.5 Results 16
4 Conclusion 21
References. 22
A.1 IPME user defined integration function 23

A.2 Listing of the cfm client code that performs the variable integration 27
A.2.1 ipme_interface.c: The module responsible for passing exchange vars with IPME 27
A.2.2 simon_integrator.c: The cfm module that processes exchange variables 35
A.2.3 cfm.h: the cfm specific header file where generic client information is defined 45
A.2.4 simon_integrator.h: header file for the client-specific functions 45
A.2.5 params.txt: 46
A.2.6 Sharedvars.txt: demonstrating the processing of one IVAR 47
A.2.7 List of abbreviations 48
Distribution list 49

List of figures

Figure 1. Standalone configuration performance 18

Figure 2. With-client configuration performance 18

Figure 3. Execution time for selected conditions as a function of the number of IVARs included in the integration routine plotted on a log scale 19

Figure 4. Execution time for selected conditions as a function of WINDUR in the integration routine, plotted on a log scale 20

List of tables

Table 1. Client-side timing variable values - simple example 6

Table 2. Client-side timing variable values – complex example 1 6

Table 3. Client-side timing variable values – complex example 2 7

Table 4. Client-side timing variable values – complex example 3 8

Table 5. Sample Params.txt file contents. The columns are: variable name, variable type, initial value 11

Table 6. Sample `sharedvars.txt` file based on one IVAR 13

Table 7. Setup of the `params.txt` file for the integration client for communications starting at 0.25 sec, repeating every 0.25 sec, using INTERVAL_HUNDREDTH. Only the first element of the first 4 lines of this file are read. Task specific notes may follow the entries on or after the first four lines 14

Table 8. Comparison of IPME simulation execution times (in seconds) 16

Table 9. Range of execution times across IVAR intervals (in seconds) 20

1 IPME and the external client

The Integrated Performance Modelling Environment (IPME) is a discrete event simulation software modelling environment that seeks to represent human behaviour in complex systems. IPME is developed and maintained by Micro Analysis & Design in Boulder, Colorado, U.S.A. It was developed as a tool for building models that simulate human and system performance. Amongst many other features, IPME contains tools for determining workload and the effects of performance shaping functions on operators.

In addition to serving as a standalone simulation suite, IPME provides mechanisms for interfacing with external models or simulations. One common use for such capability is the transfer of computationally intensive routines from the IPME task network to an external program that has been built to shoulder specific aspects of the simulations workload. The resulting architecture provides the benefit of functional modularity and enhanced performance. A block of code that is included in a precompiled external client will execute much faster than if it were embedded in IPME, where it will be interpreted at runtime. Given that the performance enhancement achieved by using an external client exceeds the slight overhead that accompanies the use of TCP/IP sockets, this approach is a favourable one for users who are interested in minimizing execution time.

This report additionally describes how the timing of client-server communication occurs with IPME clients to clarify client design as this has been found to be a conceptually difficult area. Several examples are presented to provide client programmers with insight from lessons learned. The execution time of a computationally expensive, embedded IPME function is then compared with a similar client implementation, integrating a signal over a window in time to return an average value.

2 IPME client socket communication

IPME is able to play the role of both parties in the client-server relationship. Most practical applications, including all examples used in this document, implement IPME as the server to external client applications. Similarly, most external clients developed and implemented at DRDC have been written in C and compiled for Linux using the GNU Compiler Collection (GCC).

To implement communication with a server, the client will notify the IPME server of the time of the next communication event, which becomes a registered event in the IPME event queue. Upon reaching the time designated for the next client communication event, the IPME server initiates the event and starts the communication process with the client.

During a client-aided simulation there are three main phases of client-server communication: Registration, Event Processing and Termination.

- Registration: The client identifies itself, provides names of exchange variables and any initial values that the client will be setting, and inserts the first client event into the IPME event queue.
- Event Processing: Triggered repeatedly at an interval specified in the client, each event sends the current values of those exchange variables whose value has changed since the previous event from IPME to the client. The client performs any operations implemented by the programmer, and finally passes the values of desired return variables back to the IPME server.
- Termination: At any point, either client or server can terminate the communication between them. As IPME can communicate with multiple clients simultaneously, it is important to note that terminating one client-server communication does not affect the status of any other client-server communications.

2.1 IPME sample client

2.1.1 Overview

The IPME client presented here is a customized extension of the sample IPME client provided by the manufacturer. The client uses TCP/IP sockets to exchange values of shared variables that have been updated since the last communication event; variables that have not changed since the last event are not exchanged. IPME uses port 2000 by default for the bidirectional communication but others will work. By default, the IPME server sets the initial values of the shared variables, but the client has the opportunity to overwrite these values during the initial registration process. The client enters the times of its first and subsequent events into the IPME simulation server's event queue during the initialization. The IPME simulation clock does not advance during a communication event between IPME and a client until the entire event is complete.

It is good practice to assign control of each variable to only one application, although many may use its value. Thus, if a client application assigns values to a variable, say the operator body

temperature from a thermal physiology client, then no other client or IPME should modify that variable, otherwise instabilities may occur.

2.1.2 The code

The sample IPME client code is written in the C programming language and the files are placed in the `models/CommTest/CClient/` directory of every IPME 3.x installation. The essential files required for compilation are `client.c` (or any .c file containing the main method), `common_lib.c`, `client_lib.c`, `IPME_sockets.h` and `variableTypes.h`. A Java sample client is also available.

2.1.3 Shared variables

All variables that need to be exchanged between IPME and its client must be defined by both sides. In IPME variables may be defined as global variables in the 'Define Variables/User' dialogue, as operator traits or states in the crew model, or as environment variables in the environment model. In the client, variables are defined by building a variable table (vars) that specifies the name, type and initial value (only used if client is chosen to initialize variable values) of the shared variables. The data type 'Variable' is a structure defined in 'IPME_sockets.h'. The variable names and types in both IPME and the client must be identical. An example of the client variable definition is given in Code Snippet 1 below.

Code Snippet 1. Defining 5 Client-side Shared Variables

```
char      values[5][25] = {"11", "2", "2", "2", "2"};

Variable vars[5] = {{"client_var_1",INT_TYPE, values[0]},
                    {"client_var_2",INT_TYPE, values[1]},
                    {"server_var",INT_TYPE, values[2]},
                       {"Threat.env_var",INT_TYPE, values[3]},
                       {"Operator1.oprVar.Value",INT_TYPE, values[4]}};
```
(1)

The above code snippet demonstrates the definition of 5 shared variables. In this example, the first two the variables will be modified by the client, the third will be modified by IPME as a global variable, the fourth is an IPME environment variable and the last is an IPME operator state or trait.

2.1.4 Incoming variables

Once the IPME server reaches the time of the next External Client Event in the event queue, it sends the values of each shared variable that has a different value than it had after the previous exchange. The client receives these inbound variables from the server by populating a temporary table (tmp_table). This temporary table is assessed and all the values of the updated variables that arrived from the server are identified and can be dealt with as the client program requires. The following code snippet assigns incoming variable values to their *vars* table counterpart, although

additional operations may be performed if desired. Incoming variable values will commonly be copied to local client variables, or passed into user defined client functions at this point.

Code Snippet 2.: Dealing with Incoming Exchange Variables

```
Variable_table tmp_table = NULL;
        . . .
/* get current values for interested variables */
 tmp_table = recv_variables(&num_vars, socket);
        . . .
/* process variables */
for(i = 0; i < num_vars; i++)
{
  tmp_table[i].value = (char *) ((int) tmp_table[i].value);
    strcpy (vars[i].value, tmp_table[i].value);
}
```

(2)

2.1.5 Outgoing variables

After the client has performed the desired operation using the incoming variables, it is likely that some variables will have new values to send back to IPME. When the client changes an exchange variable's value, it must update the value of the *vars* table with the updated value and change the *vars.changed* flag to true. All exchange variables flagged as having been changed will be sent from the client to IPME and their values will be applied to the corresponding variables. Additionally, the next External Client Event is scheduled in the IPME event queue. Once this is completed, the IPME simulation will resume and proceed internally until the next External Client Event time is reached.

2.2 Timing parameters

2.2.1 Configuring client event timing

When designing an IPME model, the analyst selects a time interval to correspond to the simulation clock time that is appropriate for the model. For example, modelling the tasks involved with building the pyramids, 1.0 unit of the simulation clock may represent 1 day while 10 ms may be more suited to 1 simulated time unit in a model of cognitive activity and physical response to a stimulus. When approaching the configuration of the external client's timing, one must do so in line with their model's paradigm. Two questions that are central to designing the client interface are: What are my units of time? How frequently do I need to exchange information with my external client?

2.2.1.1 Client timing variables

There are several parameters that combine to dictate the exact timing of client-sever events. All three are specified in the client code:

- **eventtime**: A long int variable. The simulation time that, when reached by the IPME server, will trigger the initial communication event with this client. This is an integer value that is modified by the prescribed time units defined below.
- **nexttime**: A long int variable. This integer value is modified by the units defined in the following variable (INTERVAL_*) to specify the interval between all subsequent communication events with this client.
- **INTERVAL_*:** An int variable, one of enum (INTERVAL_HOUR, INTERVAL_MINUTE, INTERVAL_SEC, INTERVAL_TENTH, INTERVAL_HUNDREDTH, INTERVAL_HUNDREDTH). This variable modifies the value of the *eventtime* and *nexttime* variables, influencing the way they are applied against the IPME server's simulation clock when scheduling events in the event queue. The INTERVAL refers to one whole number unit of simulation time where it is implicit that the default simulation time unit is one second.

Putting it all together, if we assume that the current simulation clock time is 10.0, and *nexttime* is 1, using INTERVAL_SEC will result in the next client event being scheduled for 11.0 by the IPME server. However, if INTERVAL_TENTH were to be specified by the client, the *nexttime* would be treated at one-tenth of a time unit, and thus the next event would be scheduled for time 10.1 in the event queue. Similarly, one INTERVAL_MINUTE would result in the next communication occurring at 70.0 simulation time units, and 3610.0 if INTERVAL_HOUR were used.

2.2.1.2 Simple example

Consider an example where a model is built using the paradigm that one IPME simulation time unit is equal to one second. The designer requires the feedback of an external client four times a second (every 250 ms). To achieve this, one can apply the following formula to determine the fractional value of each simulation time unit (one second for our current paradigm):

1.0 (simulation time unit, or second) / 4 (equal components, or 250 ms blocks per second)
= 0.25s or 25 one-hundredths of a second (3)

Since the variables *eventtime* and *nexttime* are integers, specifying their values as 0.25 with INTERVAL_SEC would not produce the desired communications. Instead, it is necessary to represent a quarter second as 25/100, which means moving our INTERVAL_* down one order of magnitude from INTERVAL_SEC to INTERVAL_HUNDREDTH.

Table 1. Client-side timing variable values - simple example

Variable Name	Value
eventtime	25
nexttime	25
INTERVAL_*	INTERVAL_HUNDREDTH

2.2.1.3 Complex example 1

Consider another model built using the paradigm that one IPME simulation time unit is equal to one year. The designer requires the feedback of an external client 52 times per year (every week). To achieve this, one can apply the following formula to determine the fractional value of each simulation time unit (one year for our current paradigm):

1.0 (simulation time unit, or year) / 52 (equal components, or weeks per year)
= 0.019230769 years (4)

The desired external client event interval is ideally 0.019230769 years, or once per week. Unfortunately the IPME external client timing protocol can only approximate this. The following values of the relevant client variables will result in a 0.02 level of granularity. To capture additional precision, it would be necessary for IPME's external client interface to support 10^{-3} granularity or for the analyst to model at a finer resolution, say 1 clock unit equates to one month or one week.

Table 2. Client-side timing variable values – complex example 1

Variable Name	Value
eventtime	2
nexttime	2
INTERVAL_*	INTERVAL_HUNDREDTH

2.2.1.4 Complex example 2

Consider a model built using the paradigm that one IPME simulation time unit is equal to one minute. The designer requires the feedback of an external client 1 time per hour (every 60 minutes). To achieve this, one can apply the following formula to determine the fractional value of each simulation time unit (one minute for our current paradigm):

1.0 (simulation time unit, or minute) * 60 (equal components, or minutes)
= 60 (5)

The desired external client event interval is ideally 60 minutes, or once per hour. Below is a table outlining the values of the relevant client variables.

Table 3. Client-side timing variable values – complex example 2

Variable Name	Value
eventtime	60
nexttime	60
INTERVAL_*	INTERVAL_MINUTE

Alternatively, although less intuitive, the analyst could have selected INTERVAL_SEC and specified eventtime and nexttime values of 3600.

2.2.1.5 Complex example 3

Consider a model built using the paradigm that one IPME simulation time unit is equal to one day. The designer requires the feedback of an external client 1 time per week (every 7 days). To achieve this, one can apply the following formula to determine the fractional value of each simulation time unit (one day for our current paradigm):

1.0 (simulation time unit, or day) * 7 (equal components, or days)
= 7 (6)

The desired external client event interval is ideally 7 days, or once per week. Below is a table outlining the values of the relevant client variables.

Table 4. Client-side timing variable values – complex example 3

Variable Name	Value
eventtime	7
nexttime	7
INTERVAL_*	INTERVAL_SEC

Although the model timing is not expressed in seconds, this is the INTERVAL_* value that will result in the *eventtime* or *nexttime* values being multiplied by 1, thus scheduling the subsequent external client event for 7 simulation time units (days) in the future. Had we chosen INTERVAL_MINUTES the *eventtime* and *nexttime* values would have been multiplied by 60, resulting in the next event being scheduled 420 simulation time units (days) in the future.

2.2.1.6 Conclusion

Rather than treating the INTERVAL_* variables based on their labels as intuition would dictate, (hours, minutes, seconds, tenths of a second, hundredths of a second) it may be more useful to treat them as 'multipliers of your modelling paradigm's time unit (n)' (n*3600, n*60, n, n/10, n/100). The programmer must then select the combination of integer *eventtime* and *nexttime* with the appropriate units of the INTERVAL_* parameter in such a way that communications between IPME and the client will occur at the simulation clock time for the established modelling time units. The IPME event queue can be used during a simulation to test the configuration by observing times of the 'External Client Events'.

2.3 A generic external client interface

2.3.1 The issue

Most of the external clients that have been used in projects at DRDC Toronto follow the same structure and in many cases maintain the same functionality, although the number and names of the exchange variables may differ. This has lead to the development of a client that was modularized, separating the IPME communication functionality from the specific operations of the client. The objective was to make one executable client program that would avoid manipulating the variable exchange code for each application yet still handle the following details in a dynamic run-time fashion:

- Specify an undefined number of shared exchange variables
- Expose the shared data for task-specific operations.
- Allow the developer to specify which variables are flagged for returning to the server.

- Allow the user to configure the timing between client and server.

2.3.2 A solution

Much of the client communication code provided with IPME has been rewritten such that it can be used with any client. This does not offer any performance enhancement or functional advantage; it merely provides standardization for easily creating clients and enhanced organization.

The Client Function Module (CFM) is a client module that plugs into the IPME external client interface module (ipme_interface.c) to provide a template for the exchange of variables between IPME and a client to which application specific functionality is added to form a complete client. The CFM may be renamed to suit any custom IPME client application, but it must include the cfm.h header file.

The CFM tracks the status of each shared variable, setting the *varStatus* array equal to 1 in the *ipme_interface.c* module at the index corresponding to the incoming variable's index in the variable table *vars*. This *varStatus* array is passed to the cfmAction() function so that the CFM can perform selective operations instead of merely brute inclusive batch operations.

Several of the client's key parameter values are read in by ipme_interface.c to increase the flexibility of the process. These parameter values may be passed along to other modules in the client *as required*. There are two files used by CFM clients to provide user defined data: *sharedvars.txt* and *params.txt*. Users may change the exchange variable names and the communication event timing through these external text files without recompiling the client.

The CFM is accessed in two ways by *ipme_interface.c* by two distinct function calls. The first function, *cfmInit()*, is called once at the beginning of the client execution to initialize the CFM variables. The second, *cfmAction()*, is called repeatedly thereafter at every scheduled client event to receive the exchange variables, process them and finally repackage them for sending back to IPME. These functions are further described below.

2.3.2.1 CFM function descriptions

1. **Initialization** is performed once, when the CFM equipped client is first executed.
   - Prototyped as: 'int cfmInit(int f_NUM_VARS, int f_outputFlag);'
   - Uses the number of shared variables to allocate memory for required operations
   - Applies user request for a CFM output file to be generated
   - Permits any other one time setup operations

2. **Action** is performed at every IPME shared variable exchange with the client.
   - Prototyped as: *'int cfmAction(Variable *f_vars, int f_NUM_VARS, int *f_varStatus, FILE *f_cfmoutputfile);'*

- Calls functions to unpack incoming shared variables, operate on them (generally by calling a custom designed function), and finally pack them up for resending to IPME via `ipme_interface.c`.
- There are 3 main components of *cfmAction()*:
  - **Unpack Variables**, called at the outset of *cfmAction()*.
    - Prototyped as: 'void unpackVars(Variable *f_vars, int f_numIvars, int *f_varStatus)'.
    - This function should be modified to organize the shared variables in a meaningful way for internal use within the CFM during the client development.
    - The programmer can choose to use *unpackVars()* to place the shared variables made available by *cfmAction* through the pointer *f_vars* into a locally defined set of structures, arrays or variables. This is the preferred programming practice. Alternatively, the variable table can be manipulated directly if so desired.
  - **User Defined Functions**
    - This comprises one or more customized routines or algorithms that are used to process the data in a meaningful way.
  - **Pack Variables**, called at the end of *cfmAction()*.
    - Prototyped as: 'void packVars(Variable *f_vars)'
    - This function should be modified to update the values of the variable table that are meant to be sent back to the IPME simulation server. It is only necessary to update the return variables that have changed and each should be flagged for return by changing its 'changed' status to *'true'*.

2.3.2.1.1 Building the client vars table

The first component of the classic IPME external client is defines the set of variables to be exchanged. Required elements include:

- the variable name (fully qualified to the Operator or model type if necessary)
- the variable type (commonly one of INT_TYPE, REAL_TYPE or STRING_TYPE Substituting the variable types with their enum values works as well: 1,2 and 7 respectively)
- the initial variable value (only used if client is configured to set initial values)

The sharedVars.txt file takes the place of this component and the structure of the file is similar to the layout of the variable definition component outlined in Appendix A of the IPME 3 User Guide (Anonymous 2004). Entries in this file may be space or tab delimited and an example of the layout is shown in Table 5. The CFM client reads the entries in the sharedVars.txt file and dynamically creates the exchange variable list at run time.

Table 5. Sample Params.txt file contents. The columns are: variable name, variable type, initial value

server_var	INT_TYPE or 1	11
client_var_1	INT_TYPE or 1	2
client_var_2	INT_TYPE or 1	2
Threat.env_var	INT_TYPE or 1	2
Operator.oprVar.Value	INT_TYPE or 1	2

2.3.2.1.2 Configuring the CFM for run time

The `ipme_interface.c` component supplied with IPME for building clients is responsible for configuring the IPME client timing and login of events. These activities have been moved outside of the code for CFM clients, into a text file called `params.txt` that is read in at run time. The `ipme_interface.c` code reads only the first element of each line, skipping all subsequent elements on the line to allow for comments. Currently, only four parameter values are read and they must be in the following order:

- *eventtime* the IPME simulation clock time of the first communication event
- *nexttime* the increments (relative to the interval used) of recurring communications
- *INTERVAL_** the "units" of the client communication times as described in section 2.2.1 (0=HOUR, 1=MIINUTE, 2=SEC, 3=TENTH, 4=HUNDREDTH)
- *Output flag* generate output file `cfmoutput` for debugging (1=YES, 0=NO)

3 Performance enhancements using clients

3.1 Overview

Increased processing efficiency may be achieved during an IPME simulation by offloading computationally demanding functions from an IPME model to an external client.

An IPME function was built to perform mathematical integration of variables over a specified window of time during a simulation and return an average value for that window to be used in subsequent IPME calculations. Simulations that used this function were found to execute slowly as the scope of the integration increased by lengthening the integration window or increasing the number of variables to be processed. A client was created to provide this functionality outside of the IPME simulation environment.

There were two main conditions in this study: 1) IPME Standalone and 2) IPME with client. Each condition comprised trials characterized by combinations of two independent variables: 1) the number of variables upon which integration was to be performed (*IVARs*), and 2) the window of time over which data points were to be integrated (*WINDUR*). For this study, the data points were generated at a regular interval so that manipulation of the WINDUR size affects the computational demand through the number of data points in the integration The expectation was that increasing either the *WINDUR* or the number of *IVARs* would increase workload, therby degrading simulation performance represented by the actual time to complete a simulation of a fixed duration. It was further expected that the time required to conduct the simulation would be longer when the integration was performed by the internal IPME function as a standalone simulation than it would in the IPME with client condition.

3.2 Setting up the ‘standalone’ mode

A simple IPME task network was constructed with a repeating task having a mean time of 0.25 with a standard deviation of 0.0 which served to generate data to test our hypotheses. Twenty global variables were defined so the maximum IVAR value is 20. Expressions were added to the beginning effects field of the repeating task that incremented all of the variable values by 1. Once each variable received its new value, each active variable was passed to the integrate function in turn. The return value from the integration function for a variable, for example *IVAR_1*, was assigned to an associated variable, for example *AvgIVAR_1*, representing the temporal average of that variable over the current WINDUR.

An IPME scenario event executed at the beginning of each simulation to set the length of the integration window, WINDUR, for each variable. The WINDUR value was held constant for the duration of each simulation and all variable had the same WINDUR value, however, WINDUR was varied between simulations.

The user defined IPME function called ‘*integrate*’ is given in Appendix A.2. The function and its parameters were defined in the following code snippet::

Code Snippet 3. IPME standalone defined function for integration

```
Float integrate(float errorVariable, float errorValue);
```
(7)

The *errorVariable* parameter is used in the standalone application to indicate to the function which variable is to be dealt with, as values are stored in arrays and accessed through the corresponding index. The *errorValue* variable contains the latest value of the variable being processed.

3.3 Setting up the 'with-client' mode

A similar IPME model was used when the simulation was conducted with a CFM integration client, instead of the IPME user defined integration function. Values for the integrated variables were generated as in the standalone mode, but the integration client returns values directly to the *AvgIVAR_** variables. The calls to the *integrate* function in the scheduling effects of the repeating task were commented out prior to executing the simulation.

An important step in porting functionality from an IPME model to an external client is variable dependency. When building a standalone model in IPME, issues of variable exposure, scope, and persistence are not complicated issues. In such a setup all user defined variables are visible throughout the model at any given time. The issue becomes more complex, however, when modularizing a system's functionality. When extracting specific functionality from an existing model, it may follow that some variables are no longer needed in the IPME server simulation and can reside solely in the client. One decision to make when designing a client / server architecture is whether or not the creation of a client would rely on an excessive amount of shared variables. In designing the integration client it became apparent that three shared variables were required for each processed variable IVAR: 1) an error, 2) a return value (measure of central tendency and 3) the WINDUR associated with the IVAR. Any information tied to the client that the server could potentially update between communication events will require a shared variable to communicate those changes to the client.

Depending upon the number of input variables, included in any particular trial, the contents of the `sharedvars.txt` file would vary. Table 6 shows that for a single IVAR, three variables are exchanged between client and server. In addition to the IVAR variables, the *clock* variable is also exchanged at each communication.

Table 6. Sample `sharedvars.txt` file based on one IVAR

Clock	REAL_TYPE	0
IVAR_error_1	REAL_TYPE	0

Avg_IVAR_1	REAL_TYPE	0
IVAR_1_WINDUR	REAL_TYPE	0

The `params.txt` file was created to create communications between IPME and the integration client every 0.25 simulation time units as shown in Table 7. Though optional, it may serve as an organizational aid to follow up each value with a contextually relevant comment and end the file with an explanation of the configuration.

Table 7. Setup of the `params.txt` file for the integration client for communications starting at 0.25 sec, repeating every 0.25 sec, using INTERVAL_HUNDREDTH. Only the first element of the first 4 lines of this file are read. Task specific notes may follow the entries on or after the first four lines

25	`# eventtime, the IPME clock time of the first communication event`
25	`# nexttime, the increments (relative to the interval used) of recurring communications`
4	`# 0=INTERVAL_HOUR, 1=INTERVAL_MINUTE, 2=INTERVAL_SEC, 3=INTERVAL_TENTH, 4=INTERVAL_HUNDREDTH`
1	# GENERATE OUTPUT FILE (`cfmoutput`) 1=YES, 0=NO

The CFM integration client structure is outlined in the following paragraphs. A listing of the code is available in Appendix **Error! Reference source not found.**.

1. *cfmInit()*: perform any initialization tasks

Place any operations that need occur only once at the outset of the program. This may include:

- allocation memory for structures, arrays, variables
- capturing configuration information for local use, ie: the *outputFlag*
- initializing variable values

For the integration client an array of *ivar* structures is dynamically allocated, the *windur* and *numinwindow* variables required for the integration procedure are initialized to 0, and the outputflag is assigned the value that was read from the params.txt file by `ipme_interface.c`.

2. *unpackVars()*: place the incoming variables appropriately for local use

This function is called by *cfmAction()* before execution of the customized function(s) in order to prepare the incoming variables for local CFM processing. The complexity of the content in *unpackVars* will vary based upon program requirements.

For the integration example there is a need for a certain amount of logic to treat all variables (after the first one, clock) as groups of three variables (error, avg, windur) that combine to characterize a single IVAR. In preparation for organizing the incoming data appropriately a structure is defined to contain all the characteristics of an IVAR (error, windur, etc…). Included in *cfmInit()* is the dynamic creation of an array of these IVAR structures based upon the number of shared variables involved in the current simulation. While looping through the entire vars table, each shared variable must be assigned to the appropriate member of the *ivar* structure array at the appropriate index for that IVAR.

This function also makes use of the varStatus array to increase efficiency of the client. The varStatus array is assigned in `ipme_interface.c` and holds a list of flags, one per shared variable, indicating whether or not they have been changed by the IPME simulation during the last exchange. It may be used to prevent unnecessary operations on unchanged variables in the client. One rule underlying the integration client was that an integration calculation is required when either an IVAR's error value OR WINDUR are changed. Thus, when unpacking the variables, an IVAR was internally flagged as 'requiring integration' when the value of varStatus for an IVAR's error value or WINDUR were set to '1'. In this way the CFM can perform operations of selective variables instead of forcing an 'en mass' batch operation on all shared variables.

3. *packVars():* identify and prepare outgoing variable for sending

This function is called by *cfmAction()* after completion of the customized function(s) in order to prepare the outgoing variables for sending to the IPME simulation server. The complexity of the logic will vary based upon program requirements. The ultimate goal is to identify those variables meant to carry values back to IPME, assign their values in a format compatible with IPME, and flag them as 'changed' so `ipme_interface.c` will send them to IPME.

In this integration client example, the information sent back to IPME is the output of the integration function for each error value / WINDUR pair, stored in the 'avgerror' member of the ivar structure. The *packVars()* function updates the values of the Avg_IVAR_* shared value with the value of that IVAR's avgerror value. A string representation of all non-string variables is required for assignment of values in the *vars* table; for REAL types, the proprietary IPME string conversion function *float2net* must be used (see common_lib.c of the IPME sample client).

4. CFMAction: establish the flow of control for the procedure

This is the function that is called from within `ipme_interface.c` so the CFM flow of control starts and ends here. A typical layout for this function is:

   a. unpackVars()

b. customized Function(s)

c. packVars()

d. return to ipme_interface

Extra logic may be included to moderate the execution of customized functions. In this integration example, a *for loop* iterates through the entire array of IVARs but only the IVARs that were identified as changed and thus flagged for integration during the unpacking phase are processed by the *integrate()* routine.

3.4 Experimental design

A set trials were run for both the 'standalone' and 'with-client' conditions using integration window durations, WINDUR values, of 1.0, 5.0, 10.0, 15.0, 20.0, 25.0 and 30.0 seconds (clock units). Each WINDUR was tested using 1, 5, 10, 15 and 20 variables to be integrated (IVARs). This setup provided 35 data point pairs that could be used to assess any performance differences between conditions.

Each trial was run for the same IPME simulation time, 300 simulation clock units. This ensured that the same number of evaluations was performed by both the standalone and with-client in any trial condition. The total amount of wall clock time, indicated in an IPME dialog upon simulation completion, was used as the dependent variable to characterise execution performance.

3.5 Results

The IPME simulation execution times for each condition are shown in Table 8. The results are expressed as the actual time required to execute the model's 300 simulation clock units. These data are also expressed as a percentage of real time where the simulation clock units are considered to represent seconds. The results are also presented graphically in Figure 1 through Figure 4.

A comparison of IPME simulation execution times between the 'standalone' and 'with-client' conditions showed marked differences in most conditions. As expected, the simulation duration increased as the number of variables to be processed (IVARs) increased. Similar trend can be observed as the integration window duration (WINDUR) increased, although in some cases, there was no apparent increase in the simulation duration to within the level of precision of the time measured.

For all but the least demanding condition, the 'with-client' configuration was able to complete the simulation much faster than the 'standalone' configuration and at a fraction of the represented time. The performance improvement of the "with-client" condition is increasingly apparent as the computational demands of the trial conditions increased, either by increasing the window over which the variables are integrated or by increasing the number of variables to integrate.

Table 8. Comparison of IPME simulation execution times (in seconds)

		Standalone		With-client	
WINDUR	**# IVARs**	**sec**	**% real time**	**sec**	**% real time**
1	1	36	12.00%	54	18.00%
	5	260	86.67%	56	18.67%
	10	514	171.33%	59	19.67%
	15	769	256.33%	68	22.67%
	20	1026	342.00%	73	24.33%
5	1	151	50.33%	54	18.00%
	5	1029	343.00%	56	18.67%
	10	2083	694.33%	62	20.67%
	15	3135	1045.00%	72	24.00%
	20	4172	1390.67%	76	25.33%
10	1	384	128.00%	55	18.33%
	5	2541	847.00%	57	19.00%
	10	4965	1655.00%	62	20.67%
	15	7568	2522.67%	75	25.00%
	20	9770	3256.67%	79	26.33%
15	1	852	284.00%	54	18.00%
	5	4437	1479.00%	59	19.67%
	10	8672	2890.67%	69	23.00%
	15	13676	4558.67%	77	25.67%
	20	18532	6177.33%	83	27.67%
20	1	1282	427.33%	54	18.00%
	5	6822	2274.00%	60	20.00%
	10	13621	4540.33%	71	23.67%
	15	21126	7042.00%	80	26.67%
	20	27511	9170.33%	90	30.00%
25	1	1864	621.33%	54	18.00%
	5	9435	3145.00%	59	19.67%
	10	19291	6430.33%	73	24.33%
	15	25972	8657.33%	85	28.33%
	20	36238	12079.33%	87	29.00%
30	1	2515	838.33%	56	18.67%
	5	12757	4252.33%	60	20.00%
	10	25761	8587.00%	76	25.33%
	15	39607	13202.33%	89	29.67%
	20	52774	17591.33%	98	32.67%

Figure 1. Standalone configuration performance

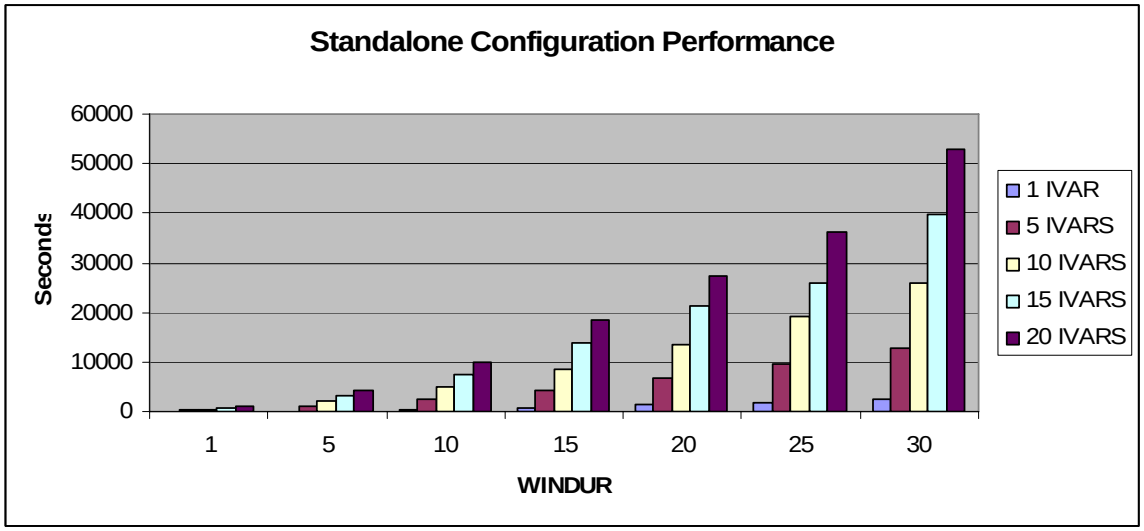
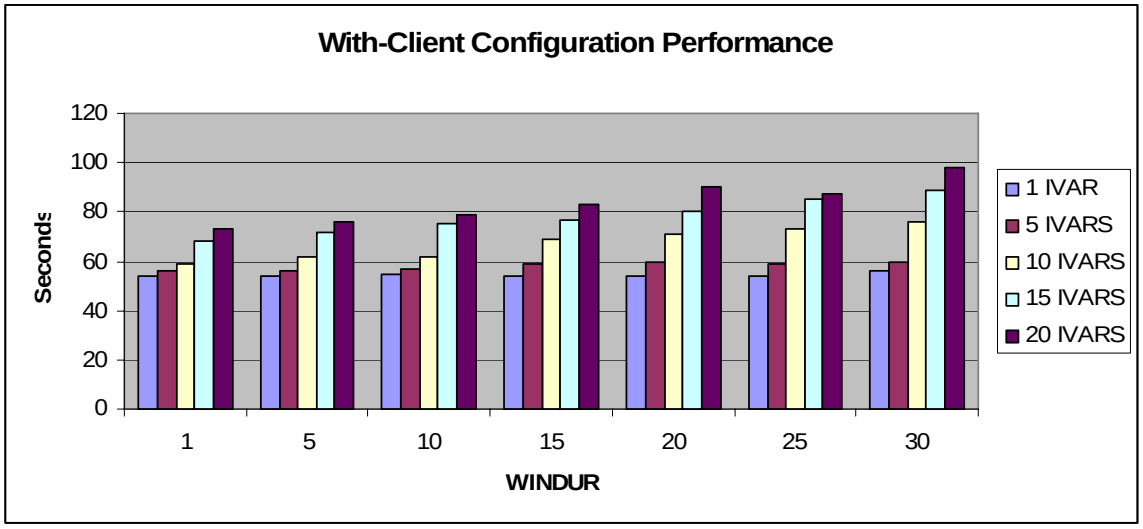


Figure 2. With-client configuration performance



There are a couple of noteworthy comparisons that reveal the superior efficiency of the 'with-client' configuration. The first is the range in execution times moving incrementally from one IVAR up to twenty. Despite the execution time being shorter for the standalone than the 'with-client' configuration when WINDUR=1 and # IVARs=1 (perhaps the impact of TCP/IP socket communication overhead), the addition of any further workload resulted in disproportionately greater performance degradation using the standalone configuration. The following table outlines

the range in execution time (seconds) across the number of IVARs ($T_{20\text{-IVARs}} - T_{1\text{-IVAR}}$) for each level of WINDUR between the 'with-client' and standalone conditions.

The execution time for the client integration routine was relatively insensitive to either the WINDUR or the number of IVARs, as shown in Figure 3 and Figure 4. Closer inspection of the data in Table 8 shows that there is a slight execution time increase for increasing either parameter, but that it is small compared to the increases observed under similar conditions with a standalone IPME user defined function. In many of the conditions, the execution time with the standalone integration function in IPME took one or two orders of magnitude longer to complete than with the integration client.

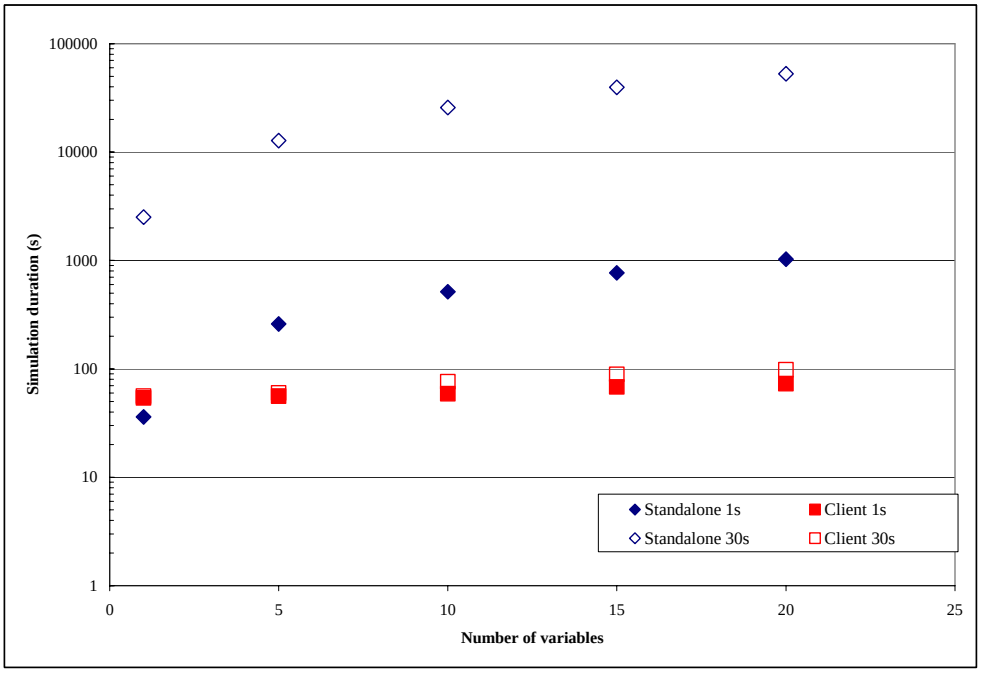


Figure 3. Execution time for selected conditions as a function of the number of IVARs included in the integration routine plotted on a log scale

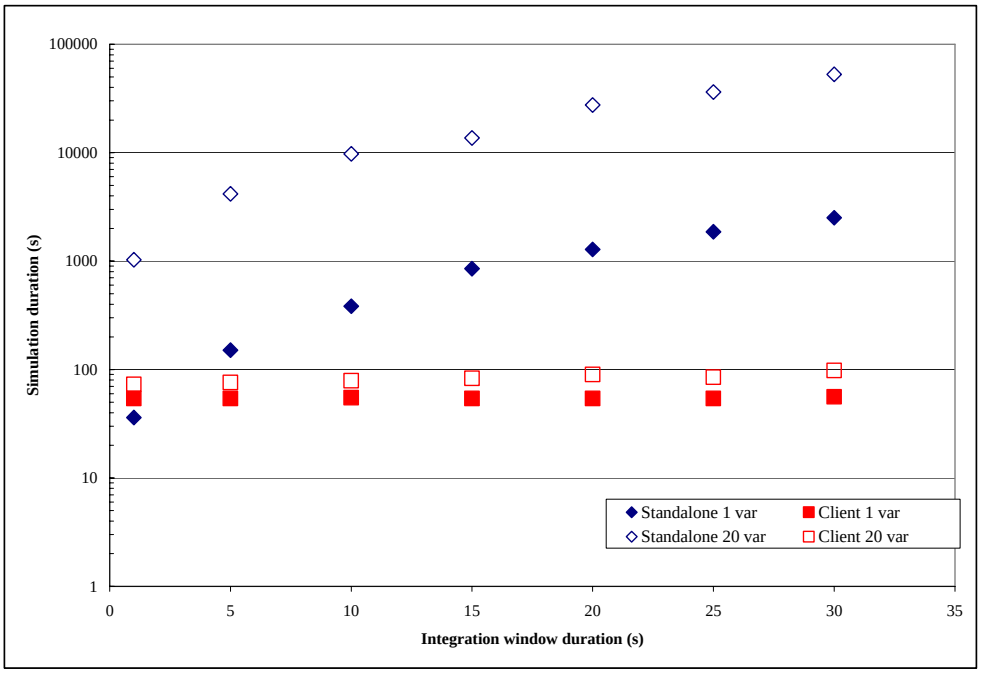


Figure 4. Execution time for selected conditions as a function of WINDUR in the integration routine, plotted on a log scale

These results indicate that when tasked to process identical workloads the 'with-client' configuration will process it in substantially less time than the standalone configuration.

Table 9. Range of execution times across IVAR intervals (in seconds)

	WINDUR						
	1	**5**	**10**	**15**	**20**	**25**	**30**
Standalone	990	4021	9386	17680	26229	34374	50259
With-client	19	21	24	29	36	33	42

Besides a remarkable difference in pure performance degradation, there was also an interesting difference concerning the way this degradation was applied. Collapsing across the number of IVARs, it was found that increasing the WINDUR caused greater degradation in the standalone condition. The 'with-client' configuration was more robust in handling a larger integration window in that it was able to 'keep up' with the heightened processing demands better than when the same calculations were processed within IPME.

4 Conclusion

IPME simulations that require computationally intensive procedures will benefit from using an external client to perform these processes rather than coding the expressions internally. Implementing an external client configuration improves processing power and speed of and IPME model. This enables the modeler to include computationally intensive procedures, and makes the model more robust to increased demand. In other words, there is small amount of overhead associated with the exchange of information between an IPME server and its client, but the benefits of increased processing power and computational robustness far outweigh this cost. A model that is supported by a client module will greatly outperform a standalone model comprising the same functionality and demand.

The Client Function Module (cfm) approach to IPME external client setup can simplify how clients are used, particularly when a versatile client is required for several applications. Building an IPME client can be somewhat daunting, but adopting practices demonstrated by the CFM can simplify the process by outlining and compartmentalizing the elements of code that need to be manipulated to arrive at the desired configuration. This reduces the amount of modification to the client code required of the end user and can eliminate the need to recompile when the client is used for similar functions in different applications.

As simulation technology improves, the potential for heightening the complexity of simulation and modeling grows with it. No longer will a modeler opt out of including certain process components in a model based on performance concerns. Scientists will be able to fill in more gaps, and replace more assumptions, or black boxes with theory and knowledge. This all leads to the exploration of more scientific questions, unhampered by technological constraint.

As simulation technology improves, the potential for heightening the complexity of simulation and modeling grows with it. Improvements to data processing power and heightened computational efficiency continue to open doors for scientists who can in turn investigate questions of greater complexity.

References

Anonymous (2004). Integrated Performance Modelling Environment. User Guide. Version 3.0. Boulder, CO., Micro Analysis and Design.

A.1 IPME user defined integration function

```
//funct PositionIntegrationWindow[int errorVariable, float errorValue]
{

/*
This function will be passed two variable values, the variable to be
controlled and the associated error. The function will pick up the time
the value of the variable was observed from the clock. Then the function
will update the error integration window and return the area under the
area-time curve for an integral control calculation.

The function makes use of a global variable for the position array that
contains all the required data for the integration components in the
control loop.

Currently, the position array is cleverly named
PositionArray[errorVariable][errorCount][errorElement]
where
errorVariable is an integer that corresponds to the current error in
question
errorCount is an index corresponding to the number of errors in the
current window, with 0 being the most recent, stretching back in time to
a numInWindow[errorVariable] up to MAX_INDEX
errorElement is an index, currently from 0:3 corresponding to Time of
the error, Error value, Error interval, Error value multiplied by Error
Interval, respectively

The funciton expects two global arrays
WINDUR[errorVariable] that contains the window in time to be used for
the integration segment - this may change in the simulation but is >=0;
and
numInWindow[errorVariable] that contains the number of relevant entries
in the error array for the integration interval

*/

float retval = 0.0; // holds the value to be returned from the function

if (WINDUR[errorVariable] > 0.0) then     // The integration window
duration is larger than zero so
                                             do some calcs else return
nothing
{
   int TIME    = 0;             // Index for time of error percept in
error array
   int ERROR   = 1;            // Index for error percept in error array
   int DELTA_T = 2;            // Index for time step duration in error
array
   int EXDT    = 3;           // Index for error*timestep_duration or
area under erorr curve at this time
```

```
   int j;                      // General purpose counters
   int k;

   int CURRENT = 0;
   int PREVIOUS = 1;

   float totalEXDT   = 0.0; // holds the quadrature estimate of the area
under the error curve segement
   float winDuration = 0.0; // active duration of integration window
which may be less than WINDUR if WINDUR was resized or at the beginning
of the sim

   /* Calculate the earliest time that can be considered for the
integration window, winLBound. The number of entries may corresponde to
an interval shorter than this if the WINDUR value is increased during
the simulation. Error values that were originally outside of the window
are excluded when the window size is recalculated and this is controlled
by the numInWndow value that can only be incremented once in any
iteration but may be decremented by any amount.
   */

   float winLBound = max(0.0, clock - WINDUR[errorVariable]);

   // Loop through the current entries in the error array, moving each
entry down one index for the
      current errorVariable

   if (numInWindow[errorVariable] > 0) then     // There is at least 1
observation of error in the
                                                   error history window
   {
         int tmp=numInWindow[errorVariable];      //tmp tracks
numInWindow for this error without
                                                     messing up the for
loop

         for (j=numInWindow[errorVariable] - 1; j >=0; j--)
         {

            /* First copy all the current values in the error array
window down one index. This will copy incorrect dt and error * dt
information from the old fist entry (index 0) into the second entry
(index 1) because the dt for the second entry will change with the
addition of the new entry, but this will be corrected below.
             */

             for (k=0; k<4; k++)
             {   PositionArray[errorVariable][j+1][k] =
PositionArray[errorVariable][j][k];      }

            // If an error occurred prior to the lower bound on the
integration window we nolonger need it or any errors that came before
it.
```

```
            if (PositionArray[errorVariable][j][TIME] < winLBound  &&
j < numInWindow[errorVariable]) then tmp--;

           }    //End for (j=numInWindow[errorVariable] - 1; j >0; j--)

            numInWindow[errorVariable] = tmp; // reset the number of
entries in the integration window

   }    //End if(numInWindow[errorVariable] > 0)

   // Add the current error to the first entry in the array

   PositionArray[errorVariable][CURRENT][TIME] = clock; // Add the
current time to the position array
   PositionArray[errorVariable][CURRENT][ERROR] = errorValue; // Add the
current error to the

position array

   //Increment the number of relevant entries in the position array to
reflect the new entry and check to make sure we haven't gone out of
bounds

   numInWindow[errorVariable]++;

   if (numInWindow[errorVariable] > MAX_INDEX) then
   {
      debug(1, MAX_INDEX, numInWindow[errorVariable], "Number of entries
error array out of bounds.");
      halt();
    }    // End if(numInWindow[errorVariable]++ > MAX_INDEX)

   // Perform integration calculations on window data


   if(numInWindow[errorVariable] > 1) then
   {

      // Compute entries for the first element of the error array

      PositionArray[errorVariable][CURRENT][DELTA_T] =
(PositionArray[errorVariable][CURRENT][TIME] -
PositionArray[errorVariable][PREVIOUS][TIME])/2.0;     // first timestep
duration
      PositionArray[errorVariable][CURRENT][EXDT] =
PositionArray[errorVariable][CURRENT][ERROR] *
PositionArray[errorVariable][CURRENT][DELTA_T];          //  first error
* duration


      // Compute entries for the last element of the error array

     int lastNum = numInWindow[errorVariable] -1;
```

```
     float dt_half = (PositionArray[errorVariable][lastNum-1][TIME] -
PositionArray[errorVariable][lastNum][TIME])/2.0;    // First half of
the timestep interval

      PositionArray[errorVariable][lastNum][DELTA_T] = dt_half +
min(dt_half, (PositionArray[errorVariable][lastNum][TIME] - winLBound));
// last timestep duration.

      /* NOTE. If WINDUR is increased during simulation, last error will
be exagerated until the
         window refills unless we limit it. My first attempt to limit
this is to limit the second
         half of the dt interval to be the smaller of either the half
time step to the next time
         or the time to the window lower bound. This may reduce the
contributions of the last error
         term to the integral, however, this effect should be minor as
the number of errors in the
         window increases.
      */

      PositionArray[errorVariable][lastNum][EXDT] =
PositionArray[errorVariable][lastNum][ERROR] *
PositionArray[errorVariable][lastNum][DELTA_T];      // last error *
duration

      /* Only need to update the second intermediate entry unless it is
the last entry, since the
         second entry contains the old first entry values from last
iteration. All other
         intermediate entries will remain the same. If the second entry
is also the last entry in the
         window, its values are calculated above.
      */

      if (numInWindow[errorVariable] > 2) then
      {
           PositionArray[errorVariable][PREVIOUS][DELTA_T] =
(PositionArray[errorVariable][CURRENT][TIME] -
PositionArray[errorVariable][PREVIOUS + 1][TIME])/2.0;    // duration
           PositionArray[errorVariable][PREVIOUS][EXDT] =
PositionArray[errorVariable][PREVIOUS][ERROR] *
PositionArray[errorVariable][PREVIOUS][DELTA_T];    // error * duration

      } //End if(numInWindow[errorVariable] > 2)


      // Calculate the area under the error curve and duration for that
error curve segment

      for (j=0; j <= numInWindow[errorVariable]; j++)
      {
           totalEXDT   += PositionArray[errorVariable][j][EXDT];
           winDuration += PositionArray[errorVariable][j][DELTA_T];
```

```
        }   // End for (j=0; j <= numInWindow[errorVariable] - 1; j++)


      //Calcualte the average value as the integral divided by the
interval for this error score

      retval = totalEXDT/winDuration;

   }   // End if(numInWindow[errorVariable] > 1)
} // End if (WINDUR[errorVariable] > 0.0)

return(retval);

} // End of func errorIntegrationWindow[]
```

A.2 Listing of the cfm client code that performs the variable integration

A.2.1 ipme_interface.c: The module responsible for passing exchange vars with IPME

```
// includes
#include <math.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <sys/time.h>
#include "IPME_sockets.h"
#include "cfm.h"
// end includes


// definitions and
#define BUFFER_LENGTH   5000
// end definitions

// variables
      int ctr, ctr_1;               // various counter needs
      int NUM_SHARED_VARS;          // number of shared vars specified
in params.txt
      char dummy[BUFFER_LENGTH];
      char typeHolder[20];
// end variables



 // main method
```

```
int main(int argc, char *argv[]){

      char * new_value;
      long eventtime;               // in seconds, 60 = 1 minute, from
params.txt
      long nexttime;                // in seconds, 60 = 1 minute, from
params.txt
      int ipmeInterval;
      int outputFlag=0;
      int done = 0;
      int i_finished = 0;
      int first_time = 0;
      FILE *paramsFile;
      FILE *sharedVarsFile;
      FILE *CFMoutputfile;

      // OPEN params.txt and extract the timing information
      paramsFile = fopen("params.txt", "r");
            fscanf(paramsFile, "%d", &eventtime); fnl(paramsFile);
            fscanf(paramsFile, "%d", &nexttime); fnl(paramsFile);
            fscanf(paramsFile, "%d", &ipmeInterval); fnl(paramsFile);
            fscanf(paramsFile, "%d", &outputFlag); fnl(paramsFile);
      fclose(paramsFile);

      Variable_table tmp_table = NULL;

  /* These are the IPME variables we are exchanging   */

  /* remember that Variable.value is a char*, not a char array
      */
  /* Thus we need to make sure there is a spot big enough to copy */
  /* variable values into                                    */
  /* you can hard code space for the variables as shown below, or */
  /* dynamiclly malloc and free space for the variables. If you have
      */
  /* complete confidence in the maximum number of characters in any
      */
  /* values being passed back and forth, the hard coding method is
fine.*/
  /* Otherwise you need to allocate the space dynamically         */

  /* I figure integers won't get more that 24 charactes long (remember)
      */
  /* to have space for a NULL byte                           */



   // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
   // Extract data from params.txt file to identify our exchange data
   // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

      // count number of parameters in parameters file
      int o;
      sharedVarsFile = fopen("sharedvars.txt", "r");
```

```
      //open to clear, close and reopend to append

      if(outputFlag){
            CFMoutputfile = fopen("CFMoutput", "w");
            fclose(CFMoutputfile);CFMoutputfile=fopen("CFMoutput", "a");
      } // END IF

      // DETERMINE NUMBER OF SHARED VARS SPECIFIED IN PARAMS.TXT
      NUM_SHARED_VARS=0;
      while(!feof(sharedVarsFile)) { fnl(sharedVarsFile);
NUM_SHARED_VARS++; }
      rewind(sharedVarsFile);       // BACK TO BEGINNING

      // CREATE OUR VARIABLE TABLE BASED ON NUM OF EXCHANGE VARS
      Variable vars[NUM_SHARED_VARS], *vars_ptr;
      int varStatus[NUM_SHARED_VARS], *varStatus_ptr;

      char valueHolder[NUM_SHARED_VARS][255]; // BUFFER FOR VAR VALUES

      // pull data into our variable table
      for(o=0;o<NUM_SHARED_VARS;o++){

            fscanf(sharedVarsFile, "%s", &vars[o].name);    // name
            fscanf(sharedVarsFile, "%s", &typeHolder);            //
type
                  if(strcmp(typeHolder, "INT_TYPE") == 0) { vars[o].type
= INT_TYPE;}
                  if(strcmp(typeHolder, "REAL_TYPE") == 0) {
vars[o].type = REAL_TYPE;}
                  if(strcmp(typeHolder, "STRING_TYPE") == 0) {
vars[o].type = STRING_TYPE;}
            fscanf(sharedVarsFile, "%s", &valueHolder[o]);  // value
                  vars[o].value = valueHolder[o];
            fnl(sharedVarsFile);    // skip to next line

      } // end for



      fclose(sharedVarsFile);

      printf("\nVARIABLES BEING EXCHANGED WITH IPME:");
      printf("\n+ + + + + + + + + + + + + + + + + + + + + + + + + + + + +
+ + + + + + + + + + + + + + + + + + + +\n");
      if(outputFlag) fprintf(CFMoutputfile, "\nVARIABLES BEING EXCHANGED
WITH IPME (THESE MUST EXIST IN IPME):");
      if(outputFlag) fprintf(CFMoutputfile, "\n+ + + + + + + + + + + + +
+ + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + +\n");

      for(o=0;o<NUM_SHARED_VARS;o++){

            printf("\n%s", vars[o].name);
            printf("\t%d", vars[o].type);
            printf("\t%s", vars[o].value);
```

```
            if(outputFlag) {
            fprintf(CFMoutputfile, "\n%s", vars[o].name);
            fprintf(CFMoutputfile, "\t%d", vars[o].type);
            fprintf(CFMoutputfile, "\t%s", vars[o].value);
            } // end if

      } // end for


      // INITIALIZE CFM
      // USE THE NUM_SHARED_VARS TO SET YOUR CFM STAGE
      if(!(cfmInit(NUM_SHARED_VARS, outputFlag))){
            printf("\nCFM initialization error\n");
            exit(0);
      } else {
      printf("\nCFM initialization success with %d shared variables.\n",
NUM_SHARED_VARS);
      }
       // END IF

      printf("\n+ + + + + + + + + + + + + + + + + + + + + + + + + + + + + +
+ + + + + + + + + + + + +\n");
      printf("\nWaiting for IPME to begin...\n");
      if(outputFlag) fprintf(CFMoutputfile, "\n+ + + + + + + + + + + + +
+ + + + + + + + + + + + +\n");


   // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *


      int num_vars;
      int i;
      int varValue;
      int socket;
      char hostName[CLIENT_NAME_SIZE];
      char clientAddr[CLIENT_NAME_SIZE];
      char clientName[CLIENT_NAME_SIZE];  /* This name must match the
client name */
                                    /* in the External Clients dialog
that  */
                                    /* comes up when The External...
button      */
                                    /* in the System Description dialog
      */

      if (argc != 3)
      {
      printf ("Usage: client IPME_Server_Node IPME_System_Name\n");
      exit (0);
      }


      gethostname (hostName, (size_t) CLIENT_NAME_SIZE);
```

```
      /* argv[0] is the full path name for this application so we need
to extract  */
      /* just the application name
      */

      for (i = strlen(argv[0]); i >= 0 && argv[0][i] != '/'; i--);
      /* back up to first /  */
      strcpy (clientName, &argv[0][++i]);                  /* copy from
just after  first /  */

      if ((socket = init_client_socket(argv[1],
                           argv[2],
                           hostName,
                           clientName,
                           clientName,
                           2000)) < 0) /* if sent addr bad */
      {
      printf ("Failed to initialize client socket\n");
      exit (-1);
      }

      sprintf (clientAddr, "%s:%s:%s", hostName, clientName, "");
      send_name(clientAddr, socket);
      printf("received time increment %d\n", recv_clock(socket));

      send_clock(ipmeInterval, socket);

      if (recv_start (socket) < 0)
      {
      printf("Panic: error receiving start message\n");
      close_client_socket(socket);
      exit(1);
      }

      /* send list of interested variables -- values are unused */
      for(int i = 0 ; i < NUM_SHARED_VARS; i++)
      {
      vars[i].changed = true;
      }
      send_variables (NUM_SHARED_VARS, vars, socket);

      /* send time of first event */
      send_time(eventtime, socket);

      /* send initial conditions */
      /* if no variable initial values are sent, the initial value
      */
      /* specified in the variable owning model is used.          */
      /* remember that initial values for client variables must be
      */
      /* initialized in this step, no initial values are set in the
      */
      /* send_variables statement above                     */
```

```
      for(int i = 0 ; i <NUM_SHARED_VARS; i++)
      {
      //vars[i].changed = true;
      }
      send_variables (0, vars, socket);     //was (2, vars, socket)

      /* tell IPME we're ready to go */
      send_wait(socket);

      while (!done)
      {


      /* wait for server to tell us it's our turn */
      for (i = 0; i < 10 && recv_go (socket) < 0; i++)

      if (i >= 10)
      {
      printf("Panic: error receiving go message\n");
      break;
      }

      //gettimeofday(&start_timeval, NULL);
      /* get current values for interested variables */
      tmp_table = recv_variables(&num_vars, socket);

      //gettimeofday(&end_timeval, NULL);
      //printf("\nTotal Time for recv_variables (microsecs): %ld\n",
(end_timeval.tv_sec-start_timeval.tv_sec) * 1000000  +
(end_timeval.tv_usec-start_timeval.tv_usec));
      //fprintf(ipmetimeoutput, "\n%lf", ((end_timeval.tv_sec-
start_timeval.tv_sec) * 1000000  + //(end_timeval.tv_usec-
start_timeval.tv_usec))/1000.0);

      if(!tmp_table && (num_vars != 0))
      {
      printf("Panic: error receiving variable values\n");
      break;
      }


      // initialize varStatus array to 0 values
      for(o=0;o<NUM_SHARED_VARS;o++)      varStatus[o] = 0;
      int varStatusIndex=0;


      // ---------------------------------------------------------------
----------
      // PROCESS VARIABLES
      /* Currently, ALL variables are checked from incoming IPME values.
Later,
            it may make sense to remove some, of which only initial
values are req'd */
```

```
            //printf("\n^^^^^ FROM client part, num_vars: %d\n",
num_vars);
            for (i=0; i< num_vars; i++)
            {

            /* variables for each ipme communication iteration */

            /* NOTE: there must be attention paid to ensure that each
vars[i] is
                  checked against the proper one as defined in vars
above. For example
                  to check for the variable as defined in vars as the
5th array element,
                  make sure you do: strcmp(tmp_table[i].name,
vars[4].name). It used to
                  compare tmp_table values against the hard coded
variable name, but
                  efforts to make the operator name dynamic required a
reference to the
                  name dynamically.               */

                  for(o=0;o<NUM_SHARED_VARS;o++){
                        if ( strcmp(tmp_table[i].name, vars[o].name) ==
0 /*&& vars[o].mode == 1*/)
                        {
                              strcpy(vars[o].value, tmp_table[i].value);
                              varStatus[o]=1; // INDICATE AN UPDATE
VALUE FOR THIS VAR
                              break;
                        } // END IF
                  } // END FOR 0<NUM_SHARED_VARS

            } // END FOR

            // As a result of the generic nature of this client, all
            // datatypes of all variables are string. It is the
programmer's
            // responsibility to convert the variables to the datatypes
            // requested by any functions that act upon them

            vars_ptr = &vars[0];
            varStatus_ptr = &varStatus[0];

            int success;

      if(!(success = cfmAction(vars_ptr, NUM_SHARED_VARS, varStatus_ptr,
CFMoutputfile))){
                  printf("\nfunction 'clientFunctionModuleAction' failed
to complete. Exiting...\n");
                  exit(0);
            } // END IF ! success

      // In this model, the updating of vars table is done in the
'client function
```

```
      // module (CFM)'. All that needs to be done now is the sending.
All return
      // vars have been populated and flagged as changed=true in the
'packVars' function.

       // ---------------------------------------------------------------
-----------
       // Step 11: Sending IPME an event completed signal
          send_event_complete(socket);

       // ---------------------------------------------------------------
-----------
       // Step 12: Send variables that is changed in this process */
          send_variables(NUM_SHARED_VARS, vars, socket);

       // ---------------------------------------------------------------
-----------
       // delete tmp_table
          if (tmp_table != NULL)
         {  delete tmp_table;
         }

       // ---------------------------------------------------------------
-----------
       // Step 13: Send time of next event
          eventtime += nexttime;
          send_time(eventtime, socket);

       // ---------------------------------------------------------------
-----------
       // Step 14: Send signal to IPME to indicate that the client is
ready for next process.
          send_wait(socket);
   }

  close_client_socket(socket);


      return 0;

} // end main







/*
  these are required functions for the common library
*/
void ServerExitCallback(int s)
{
```

```
  close_client_socket(s);
  printf("Exit received from server -- exiting\n");
  fflush(stdout);
  exit(1);
}

void ServerAbortCallback(int s)
{
  close_client_socket(s);
  printf("Abort received from server -- exiting\n");
  fflush(stdout);
  exit(1);
}


void ServerCompleteCallback(int s)
{
  close_client_socket(s);
  printf("Complete received from server\n");
  fflush(stdout);
  exit(1);
}

void ServerPauseCallback(int s)
{
  printf("Pause Received from server\n");
  fflush(stdout);
}

void ServerResumeCallback(int s)
{
  printf("Resume Received from server\n");
  fflush(stdout);
}

// MOVES THE CURSOR TO THE NEXT LINE IN THE FILE
void fnl(FILE *f){
   fgets(dummy,BUFFER_LENGTH,f);
  }
```

A.2.2 simon_integrator.c: The cfm module that processes exchange variables

```
#include "cfm.h"
#include "simon_integrator.h"

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* This function works under 2 assumptions:
* 1) the variables in the vars parameter are consistent with the
```

```
* variables laid out in params.txt
* 2) Any return variables are flagged as 'changed' within this
* code, not in the IPME socket interface.
* This system makes for a more generic IPME socket variable exchange
* interface.
*
* As a 'client function module' (cfm) this receives a pointer to the
* first element of 'vars', which is an array of type Variable_table,
* which is a pointer type of Variable. This gives the function access
* to the raw data. Since each data member's value is in char format,
* translation of required variables to their desired type is to be
* done here. Once the value of returning variables have been updated,
* the re-translation into char is done here as well. This again, makes
* for a more generic IPME socket variable exchange interface as it is
* now only responsible for populating the vars table and sending back
* all variables that have 'changed=true' after the function module has
* completed it's task.
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

// define the local variables that will hold copies of our exchange
// variables to be manipulated by our 'cfm'.
// The variables here are dictated by the needs of the cfm, and are
// directly associated with the variables outlined in params.txt
float clock;

ivar *ivars;

FILE *l_CFMoutputfile;

float       winDuration;            // amount of time window spans
float       winLBound;        // adjusts to time of newX minus
winDuration
float       winUBound;        // same as newX time, for readability
int   numInWindow;            // number of err scores in the window
float       totalError;       // sum of error over window
float       totalDeltaT;            // sum of deltaT over window
float       totalXDT;         // sum of (errorXdeltaT) over window
float       currentTime;                  // to hold the most recent
clock time from IPME
float       retval;                 // final answer
int   numIvars;         // how many variables we want to perform
integration on
                        // this will define the number of ivar packs we
use
int NUM_VARS;                 // number of total exchange vars,
including clock
int outputFlag;
int integrateVarsStart;       // index of the first element of the first
3 pack
int NUM_IN_PACK;        // Exchange vars are in groups of 3
float ErrorArray[MAX_IVARS][MAX_INDEX][4];      // array to hold desired
err vals and times
                        // NOTE: only storing calc steps (delta_T,
errorXdeltaT)
```

```
                        // for display purposes.
                        // for efficiency, we could simply accumulate
running
                        // totals in 1 float.




// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
// This fires only once at the beginning of the client's life
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
int cfmInit(int f_NUM_VARS, int f_outputFlag){
      int i;
      ivars = (ivar *)malloc(f_NUM_VARS * sizeof(ivar)); // allocate
memory
      for(i=0;i<f_NUM_VARS;i++) {
            ivars[i].numInWindow=0;
            ivars[i].windur=0.0;
      } // end for
      outputFlag = f_outputFlag;
      return 1;

} // end cfmInit()



/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* define your customized function here that was declared in
* cfm.h and called in cfmAction();
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
float integrate(float f_currentTime, int ivarIndex){

      printf("\n\n-----------------------------------------------------
");
      printf("\nINTEGRATING FOR IVAR AT INDEX %d\n\n", ivarIndex);
      printf("\n\n-----------------------------------------------------
");

      if(outputFlag){
            fprintf(l_CFMoutputfile, "\n\nINTEGRATING FOR IVAR AT INDEX
%d", ivarIndex);
            fprintf(l_CFMoutputfile, "\n-------------------------------
--------------------\n");
      } // END IF

      // ASSOCIATE WITH THE CORRECT ERRVAR'S INFORMATION
      numInWindow = ivars[ivarIndex].numInWindow;

      // GET THE WINDOW UPPER AND LOWER LIMITS
      winUBound = f_currentTime;

      winDuration = ivars[ivarIndex].windur;
```

```
      // if Ubound is less than defined WINDUR, make winDuration = to
winUBoud
      //if(winDuration<WINDUR) winDuration = winUBound < WINDUR ?
winUBound : WINDUR;

      winLBound = (winUBound - winDuration)< 0.0 ? 0.0 : (winUBound -
winDuration);
      totalDeltaT=0.0; // REFRESH IT
      totalError=0.0;
      totalXDT=0.0;

      // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * *
      // IF THERE ARE EXISTING RECORDS IN THIS IVAR'S "MEMORY" WINDOW
      // WE MUST SHIFT THEM ALL DOWN ONE
      // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * *

      if(numInWindow > 0){

            int tmp=numInWindow; // tmp TRACKS numInWindow WITHOUT
MESSING UP THE FOR LOOP
            int j,k;

            // LOOP THROUGH CURRENT ErrorArray FOR THIS IVAR. MOVE EACH
SCORE DOWN ONE INDEX.
            for(j=numInWindow-1;j>=0;j--){

                  for(k=0;k<4;k++)
ErrorArray[ivarIndex][j+1][k]=ErrorArray[ivarIndex][j][k];

                  // IF A SCORE HAS MOVED PAST THE WINDOW, DECREASE
                  // numInWindow BY ONE. WE NO LONGER NEED THAT SCORE.
                  if(ErrorArray[ivarIndex][j][TIME]< winLBound &&
j<numInWindow) tmp--;

            } // END FOR

            // APPLY NEW numInWindow
            numInWindow=tmp;

      } // END IF

      // LOG CURRENT ERROR VALUE THAT IS BEING ADDED TO TOP OF WINDOW

      // TIME TO POPULATE THE FIRST INDEX WITH THE NEW ERRORVAL
      ErrorArray[ivarIndex][CURR][TIME]=f_currentTime;
      ErrorArray[ivarIndex][CURR][ERRORVAL]=ivars[ivarIndex].error;

      // SINCE WE HAVE AT LEAST ONE RECORD, REFLECT IT IN numInWindow
      numInWindow++;

      // OUTPUT TO FILE START
```

```
      if(outputFlag) fprintf(l_CFMoutputfile, "\nWORKING ON ERROR VALUE
OF: %f\n", ErrorArray[ivarIndex][CURR][ERRORVAL]);

      int t, LAST;
      float dt_half,dt_minus_wlbound;


      // IF WE HAVE MORE THAN ONE (OR AT LEAST ONE) ERROR VALUE IN
WINDOW...
      if(numInWindow>1){

            // PERFORM INTEGRATION CALCULATIONS ON WINDOW DATA

                  // IF IT IS THE FIRST ONE...

                  // CALCULATE DT

      ErrorArray[ivarIndex][CURR][DT]=(ErrorArray[ivarIndex][CURR][TIME]
-ErrorArray[ivarIndex][PREV][TIME])/2.0;

                  // MULTIPLY ERRORVAL by DT
                  ErrorArray[ivarIndex][CURR][XDT] =
ErrorArray[ivarIndex][CURR][ERRORVAL] * ErrorArray[ivarIndex][CURR][DT];


                  // IF IT'S THE LAST ONE WE APPLY A DIFFERENT FORMULA
                  LAST = numInWindow-1;

                  // CALCLULATE FIRST HALF OF LAST TIME INTERVAL
                  dt_half = (ErrorArray[ivarIndex][LAST-1][TIME] -
ErrorArray[ivarIndex][LAST][TIME]) / 2.0;
                  dt_minus_wlbound = ErrorArray[ivarIndex][LAST][TIME]-
winLBound;
                  ErrorArray[ivarIndex][LAST][DT]= dt_half;
                  //ErrorArray[ivarIndex][LAST][DT]+= dt_half <
dt_minus_wlbound ? dt_half : dt_minus_wlbound;


                  // NOTE. If WINDUR is increased during simulation,
last error will be exagerated until the window refills unless we limit
it. My first attempt to limit this is to limit the second half of the dt
interval to be the smaller of either the half time step to the next time
or the time to the window lower bound. This may reduce the contributions
of the last       error term to the integral, however, this effect
should be minor as the number of errors in the window increases.
                  ErrorArray[ivarIndex][LAST][XDT] =
ErrorArray[ivarIndex][LAST][ERRORVAL] * ErrorArray[ivarIndex][LAST][DT];

                  // Only need to update the second intermediate entry
unless it is the last entry, since the second entry contains the old
first entry values from last  iteration. All other intermediate entries
will remain the same. If the second entry is also the last entry in the
window, its values are calculated above.
```

```
                // IF WE HAVE AT LEAST 2 ERRORS IN OUR WINDOW
                if(numInWindow > 2) {

                      // DURATION
                      ErrorArray[ivarIndex][PREV][DT] =
(ErrorArray[ivarIndex][CURR][TIME]-ErrorArray[ivarIndex][2][TIME])/2.0;
                      // ERRORVAL * DURATION
                      ErrorArray[ivarIndex][PREV][XDT] =
ErrorArray[ivarIndex][PREV][ERRORVAL] * ErrorArray[ivarIndex][PREV][DT];

                } // END IF numInWindow > 2


                // CALCULATE THE AREA UNDER THE ERRORVAL CURVE AND
DURATION FOR THAT ERRORVAL CURVE SEGMENT
                int s;
                for(s=0; s<=numInWindow-1; s++){

                      totalXDT += ErrorArray[ivarIndex][s][XDT];

                } // END FOR

                // UPDATE TOTALS USED IN retval CALCULATION
                // & PRINT OUT THE CURRENT STATUS OF THE ErrorArray
                for(t=0;t<numInWindow;t++){

                      totalDeltaT+=ErrorArray[ivarIndex][t][DT];
                      totalError+=ErrorArray[ivarIndex][t][ERRORVAL];
                      // OUTPUT TO FILE START
                      if(outputFlag) fprintf(l_CFMoutputfile,
"\nwa[%d]\tivarIndex: %d\terr: %lf\t\ttime: %lf\t\tdt: %lf\t\teXdt:
%lf",
                      t, ivarIndex,
ErrorArray[ivarIndex][t][ERRORVAL], ErrorArray[ivarIndex][t][TIME],
ErrorArray[ivarIndex][t][DT], ErrorArray[ivarIndex][t][XDT]);

                } // END FOR


                // COMPUTE RETURN VAL (INTEGRATED VALUE BASED ON
ERRORVAL AND WINDOW VALS
                retval = totalXDT/totalDeltaT;

                // OUTPUT TO FILE START
                if(outputFlag){
                fprintf(l_CFMoutputfile, "\n--");
                fprintf(l_CFMoutputfile, "\n\tTerr: %lf\t\t\t\t\tTdt:
%lf\t\tTXDT: %lf\t\tnumInWindow: %d",totalError, totalDeltaT, totalXDT,
numInWindow);
                fprintf(l_CFMoutputfile, "\n-------------------------
------------------------------------------------------------------------
-");
                fprintf(l_CFMoutputfile, "\n\tretval: %lf", retval);
                fprintf(l_CFMoutputfile, "\n------\n\n\n");
```

```
                  } // end if

      } else {// WE ONLY HAVE ONE RECORD, SO WE'LL RETURN THE INDIVIDUAL
ERROR AS OUR AVGERROR

            retval = ErrorArray[ivarIndex][CURR][ERRORVAL];

      } // END IF numInWindow > 1


      // LOG THE numInWindow so we can pick up where we left off next
time for this ivar
      ivars[ivarIndex].numInWindow = numInWindow;

      return retval;

} // END integrate



// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
// This is the action function where the specific functionality
// happens. This function fires once for each IPME socket cycle
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
int cfmAction(Variable *f_vars, int f_NUM_VARS, int *f_varStatus, FILE
*f_CFMoutputfile){

      //printf("\n+ + + + + + + IN CFM: + + + + + + + + ");
      //printf("\n+ + + THESE VARS CAME OVER + + + + + ");
      int a;
      //for(a=0;a<f_NUM_VARS;a++) if(f_varStatus[a] >0) printf("\n@ @ @
@ @ @ @ @ @ @ >>> f_varStatus[%d]= %d <<< @ @ @ @ @ @ @ @ @ @ @ @",
a,f_varStatus[a]);

      l_CFMoutputfile = f_CFMoutputfile;

      if(outputFlag) fprintf(l_CFMoutputfile, "\n\nNEW ROUND OF
INTEGRATION\n+ + + + + + + + + + + + + + + + + + + + + + + + + + + + + +
+ + + + + + + + ");

      NUM_VARS = f_NUM_VARS;
      int errorIndex;
      integrateVarsStart = 1;
      NUM_IN_PACK = 3;

      int ivarCheck = (NUM_VARS-1)%NUM_IN_PACK;
      if(ivarCheck!=0){ printf("\nExchange Variables out of sync!\n");
exit(1); }
      numIvars = (NUM_VARS-1) / 3;

      // COPY EXCHANGE VARS TO LOCAL COPIES
      // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * * * * * * *
      unpackVars(f_vars, f_varStatus);
```

```
	// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

	for(a=integrateVarsStart; a <= (NUM_VARS - integrateVarsStart); a+=NUM_IN_PACK){

		//printf("\n%s.status: %d", f_vars[a].name, ivars[a].status);

	} // end for


	// CALL CFM SPECIFIC FUNCTION HERE
	// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
	if(outputFlag) fprintf(l_CFMoutputfile, "\n\nSTARTING INTEGRATION ON NEW EXCHAGE VARS (clock: %f | numIvars: %d\n", clock, numIvars);

	for(errorIndex=0; errorIndex < numIvars; errorIndex++){

		if(ivars[errorIndex].status){

			//printf("\nINTEGRATING ON: ivar with index %d (windur: %f)\n", errorIndex, ivars[errorIndex].windur);
			ivars[errorIndex].avgerror = integrate(clock, errorIndex); // CALL integrate FUNCTION FOR EACH EXCHANGE VAR
		} // END IF STATUS IS '1'

	} // END FOR

	// OUTPUT TO FILE
	if(outputFlag) fprintf(l_CFMoutputfile, "\nINTEGRATION COMPLETE\n");
	for(errorIndex=0; errorIndex < numIvars; errorIndex++){
		if(ivars[errorIndex].status){
			if(outputFlag) fprintf(l_CFMoutputfile, "\tivars[%d].avgerror = %f\n", errorIndex, ivars[errorIndex].avgerror);
		}
	} // END FOR

	// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *



	// PACKAGE VARS FOR RETURN
	// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
	packVars(f_vars);
	// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

```
      return 1;

} // end clientFunctionModuleAction



// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
// Utility Functions
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
// here you populate the cli-scope variables using the vars table
// that came over from the ipme client interface. We are only
// interested in the IPME updated variables here.
// This is specific to each implementation of the CFM.
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
void unpackVars(Variable *f_vars, int *f_varStatus){

      //printf("\nstarting unpack vars\n");


      int i,j,current_pack,current_ivar, integration_needed;

      // RETRIEVE clock, THE ONLY EXCHANGE VAR NOT IN A PACK (ie: VARS
THAT WORK TOGETHER)
      clock = atof(f_vars[0].value);


      // since the data comes in 3 packs (error, avgerror, windur)
      // I'm putting each set in a struct. The number of elements in my
      // array of ivars will be the total number of exchange variables
      // minus 1 (clock) divided by 3.
      // For any given integration request, there will be varying
members
      // of the ivars array that require integration, as signified by an
      // update to their value (either the error value or the WINDUR)
      // since the last integration cycle.
      current_pack=0;


      //printf("\nJust created ivar for this round, made room for %d
ivars.\n", numIvars);

      // LOOP THROUGH IVARS AND ACT UPON ALL WHOSE STATUS IS '1'
      // INDICATING THEY'VE BEEN UPDATED AND REQUIRE A ROUND OF
INTEGRATION
      for(i=integrateVarsStart; i <= (NUM_VARS - integrateVarsStart);
i+=NUM_IN_PACK){
            //printf("\nAnalysing ivar: %s for INTEGRATION
requirement...", f_vars[i].name);
            integration_needed=0;
            // create my array of ivars
            // populate local vars from exchange table
```

```
            if(f_varStatus[i]>0) { ivars[current_pack].error =
atof(f_vars[i].value); integration_needed=1; /* printf("\nSET %s error
to: %f\n", f_vars[i].name, ivars[current_pack].error);*/}
            //ivars[current_pack].avgerror = 0.0;
//atof(f_vars[i+1].value);
            // IF THE WINDUR CHANGES, THE STATUS SHOULD CHANGE TO SIGNAL
FOR AN INTEGRATION
            if(f_varStatus[i+2]>0) { ivars[current_pack].windur =
atof(f_vars[i+2].value); integration_needed=1; /* printf("\nSET %s
windur to: %f\n", f_vars[i+2].name, ivars[current_pack].windur); */}

            if(integration_needed) ivars[current_pack].status = 1; else
ivars[current_pack].status = 0;

            //printf(" ... ivars[%d].status is %d", current_pack,
ivars[current_pack].status);
            //printf(" ... ivars[%d].status is %d", current_pack,
ivars[current_pack].status);
            current_pack++;

      } // end for j

} // END unpackVars


// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
// Here you repopulate the returning vars values with the
// CFM values and flag them 'changed = true'
// This is specific to each implementation of the CFM.
// 'float2net' IS USED FOR THE BENEFIT OF IPME
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
void packVars(Variable *f_vars){

      int i;
      int ivar_index=0;

      for(i=integrateVarsStart;i<=(NUM_VARS-
integrateVarsStart);i+=NUM_IN_PACK){
            //printf("\n\n\nPACKVARS:\n\tintegrateVarsStart: %d\n\ti:
%d\n\tivars[%d].avgerror: %f\n",
            //integrateVarsStart,i,ivar_index,
ivars[ivar_index].avgerror);

            if(ivars[ivar_index].status){

                  f_vars[i+1].value =
float2net(ivars[ivar_index].avgerror);
                  f_vars[i+1].changed = true;

            } // end if

            ivar_index++; // mode ivar index up to next one

      } // END FOR
```

```
} // END packVars
```

A.2.3 cfm.h: the cfm specific header file where generic client information is defined

```
#if defined(CFM_HEADER)
/* the file has been included already */
#else
#define CFM_HEADER

// includes
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "IPME_sockets.h"


// prototypes
int cfmInit(int f_NUM_VARS, int f_outputFlag);
int cfmAction( Variable *vars,
                        int f_NUM_VARS,
                        int *f_varStatus,
                        FILE *f_CFMoutputfile);
void packVars(Variable *vars);
void unpackVars(Variable *vars, int *f_varStatus);
void fnl(FILE *f);


#endif
```

A.2.4 simon_integrator.h: header file for the client-specific functions

```
#if defined(TRACKER_INTEGRATOR)
/* the file has been included already */
#else
#define TRACKER_INTEGRATOR

// includes
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * */
// CLIENT SPECIFIC CODE BELOW : that which applies to THIS CFM
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * */

// defines
#define MAX_INDEX       10000
// number of historical values possible in an integration window
#define MAX_IVARS 50
                        // number of variables that will require
integration (henceforth 'ivar')
                        // and therefore their own numInWindow, winDur
values
#define CURR      0           // current error index (most recent) in
the integration window
#define PREV      1           // previous error index (2nd most recent)
in the integration window

// prototypes
float integrate(float f_currentTime, int ivarIndex);

// structures
typedef struct integrateVar{
      float error;
      float windur;
      int numInWindow;
      float avgerror;
      int status;
} ivar;

// enums
enum {TIME, ERRORVAL, DT, XDT};     // 0:time, 1:coordinate, 2:deltaT,
3:extended deltaT (deltaT*coordinate)

#endif
```

A.2.5 params.txt:

```
3     # eventtime, the ipme clock time of the first communication event
3     # nexttime, the increments (relative to the interval used) of
recurring communications
4     # 0=INTERVAL_HOUR, 1=INTERVAL_MINUTE, 2=INTERVAL_SEC,
3=INTERVAL_TENTH, 4=INTERVAL_HUNDREDTH
1     # GENERATE OUTPUT FILE (`CFMOUTPUT`) 1=YES, 0=NO
```

```
# NOTE: Only the first 4 lines of this file are read. Task specific
notes may follow
- Starting at 0.25 sec, repeating every 0.25 sec, using
INTERVAL_HUNDREDTH
```

A.2.6 Sharedvars.txt: demonstrating the processing of one IVAR

```
clock                              REAL_TYPE   0.0
Ivan.Perceived.AltitudeError       REAL_TYPE   0.0
Ivan.Perceived.AvgAltitudeError    REAL_TYPE   0.0
Ivan.Perceived.AltitudeWINDUR      REAL_TYPE   0.0
```

A.2.7 List of abbreviations

CFM	client function module
IPME	Integrated Performance Modelling Environment
IVAR	The number of variables to be processed by the client and exchanged between the client and IPME
WINDUR	The window of time over which the change in a variable is to be calculated using mathematical integration

Distribution list

Document No.: DRDC Toronto TM 2007-033

LIST PART 1: Internal Distribution by Centre:

1 Internal Library
2 Mr. Phil ter Haar
2 Mr. Brad Cain

5 TOTAL LIST PART 1

LIST PART 2: External Distribution by DRDKIM

1 DRDKIM
1 National Archives

2 TOTAL LIST PART 2

7 TOTAL COPIES REQUIRED

DOCUMENT CONTROL DATA
(Security classification of the title, body of abstract and indexing annotation must be entered when the overall document is classified)

1. ORIGINATOR (The name and address of the organization preparing the document, Organizations for whom the document was prepared, e.g. Centre sponsoring a contractor's document, or tasking agency, are entered in section 8.)

Publishing: DRDC Toronto
Performing: DRDC Toronto
Monitoring:
Contracting:

2. SECURITY CLASSIFICATION
(Overall security classification of the document including special warning terms if applicable.)

UNCLASSIFIED

3. TITLE (The complete document title as indicated on the title page. Its classification is indicated by the appropriate abbreviation (S, C, R, or U) in parenthesis at the end of the title)

IPME and external clients: Enhancing performance by offloading simulation workload to external clients; explaining and simplifying the process (U)
EIMP et clients externes Amélioration des performances par délestage de la charge de travail à des clients externes : explication et simplification du processus (U)

4. AUTHORS (First name, middle initial and last name. If military, show rank, e.g. Maj. John E. Doe.)

Phil ter Haar; Brad Cain

5. DATE OF PUBLICATION (Month and year of publication of document.)	6a NO. OF PAGES (Total containing information, including Annexes, Appendices, etc.)	6b. NO. OF REFS (Total cited in document.)
December 2007	51	1

7. DESCRIPTIVE NOTES (The category of the document, e.g. technical report, technical note or memorandum. If appropriate, enter the type of document, e.g. interim, progress, summary, annual or final. Give the inclusive dates when a specific reporting period is covered.)

Technical Memorandum

8. SPONSORING ACTIVITY (The names of the department project office or laboratory sponsoring the research and development − include address.)

Sponsoring:
Tasking:

9a. PROJECT OR GRANT NO. (If appropriate, the applicable research and development project or grant under which the document was written. Please specify whether project or grant.)	9b. CONTRACT NO. (If appropriate, the applicable number under which the document was written.)
16BR02	
10a. ORIGINATOR'S DOCUMENT NUMBER (The official document number by which the document is identified by the originating activity. This number must be unique to this document)	**10b. OTHER DOCUMENT NO(s). (Any other numbers under which may be assigned this document either by the originator or by the sponsor.)**
DRDC Toronto TM 2007−033	

11. DOCUMENT AVAILABILITY (Any limitations on the dissemination of the document, other than those imposed by security classification.)

Unlimited distribution

12. DOCUMENT ANNOUNCEMENT (Any limitation to the bibliographic announcement of this document. This will normally correspond to the Document Availability (11), However, when further distribution (beyond the audience specified in (11) is possible, a wider announcement audience may be selected.))

Unlimited announcement

DOCUMENT CONTROL DATA
(Security classification of the title, body of abstract and indexing annotation must be entered when the overall document is classified)

13. ABSTRACT (A brief and factual summary of the document. It may also appear elsewhere in the body of the document itself. It is highly desirable that the abstract of classified documents be unclassified. Each paragraph of the abstract shall begin with an indication of the security classification of the information in the paragraph (unless the document itself is unclassified) represented as (S), (C), (R), or (U). It is not necessary to include here abstracts in both official languages unless the text is bilingual.)

(U) Performance of the Integrated Performance Modeling Environment (IPME) degrades significantly when computationally demanding functions are included in a model. Much of this degradation can be mitigated by transferring the resource intensive functionality to an external client using the external client architecture that accompanies IPME. This client communicates with IPME using TCP/IP network protocol, exchanging values of common variables over the network. Using an external client allows more processing power to be dedicated to the computationally expensive task. It is also more robust to increases in the demands on these tasks. A computationally demanding sample client is used to show the execution performance differences when the procedure resides within the IPME task network and when it is offloaded to an external client. This report also outlines how to use the sample client source code to build a client program, extending the developer's approach for client program development to create a more flexible interface.

(U) Les performances de l'environnement intégré de modélisation des performances (EIMP) se dégradent considérablement lorsque des fonctions exigeant beaucoup de puissance de calcul sont intégrées dans un modèle. Cette dégradation peut être atténuée considérablement en transférant la fonction exigeant beaucoup de puissance à un client externe qui utilise l'architecture de client externe connexe à l'EIMP. Ce client communique avec l'EIMP au moyen du protocole réseau TCP/IP, échangeant ainsi des valeurs de variables partagées par le réseau. L'utilisation d'un client externe permet de réserver plus de puissance de traitement à la tâche exigeant beaucoup de puissance de calcul. Cela permet aussi de mieux répondre aux augmentations de la demande visant ces tâches. Un exemple de client exigeant beaucoup de puissance est utilisé pour faire la démonstration de la différence sur le plan des performances à l'exécution entre une procédure résidant dans le réseau d'exécution de tâche de l'environnement EIMP et la même procédure transférée à un client externe. Ce rapport décrit aussi comment utiliser le code source de l'exemple de client pour produire un programme client, ce qui étend la portée de la démarche du développeur en matière de développement de programmes clients afin de pouvoir créer une interface plus souple.

14. KEYWORDS, DESCRIPTORS or IDENTIFIERS (Technically meaningful terms or short phrases that characterize a document and could be helpful in cataloguing the document. They should be selected so that no security classification is required. Identifiers, such as equipment model designation, trade name, military project code name, geographic location may also be included. If possible keywords should be selected from a published thesaurus, e.g. Thesaurus of Engineering and Scientific Terms (TEST) and that thesaurus identified. If it is not possible to select indexing terms which are Unclassified, the classification of each should be indicated as with the title.)

(U) IPME; external clients; performance enhancement; human modeling; moderator; cfm; performance prediction; TCP/IP Socket Communication

Defence R&D Canada

Canada's Leader in Defence
and National Security
Science and Technology

R & D pour la défense Canada

Chef de file au Canada en matière
de science et de technologie pour
la défense et la sécurité nationale

DEFENCE DÉFENSE

www.drdc-rddc.gc.ca