A computational framework for flood risk assessment in the Netherlands

A.A. Markus[1], W.M.G. Courage[2] and M.C.L.M. van Mierlo[3]

1 Deltares[1] (unit Deltares Software Centre), Delft, The Netherlands, corresponding author[2]
2 TNO Built Environment and Geosciences, Delft, The Netherlands
3 Deltares (unit Inland Water Systems), Delft, The Netherlands

Abstract

The safety of dikes in the Netherlands, located in the delta of the rivers Rhine, Meuse and Scheldt, has been the subject of debate for more than ten years. The safety (or flood risk) of a particular area may depend on the safety of other areas. This is referred to as effects of river system behaviour on flood risk (quantified as the estimated number of casualties and economic damage). In the Netherlands, such effects are found to be important for the understanding and evaluation of flood risks.

A computational framework was developed to assess these effects. One component in this system selects reasonable dike strength parameters (based on the measured strength parameters and their uncertainty) and flow conditions via a Monte Carlo simulation. A second computes maps of the water level and flow velocities in the study area as a consequence of a dike breach. The third and final component uses these maps and maps of the population density and economic activities to estimate the potential number of casualties and the economic damage.

Flood risk is in and by itself a probabilistic problem: it depends on stochastic parameters such as the actual hydraulic load (the flood event) and the strength of dike sections, which is known only approximately. To handle this aspect, we used a Monte Carlo method to select 81 computations for which dike breaches would occur given the hydraulic and strength characteristics, out of a total of seven million. Preparing the input data, running the various computations and getting the final results was automated using Tcl scripts as much as possible. The hydraulic computations would last several days and were conducted in parallel on a Linux cluster to keep the time required within reasonable limits.

Though this complex system was set up *ad hoc*, we can describe it more formally as a *tuplespace*, each computational step for each of the 81 computations thus is represented as a unique *tuple*, a combination of job ID and status. Within the Linux cluster each processor was assigned one such tuple, so that the next step could be carried. The tuplespace paradigm allows one to set up arbitrarily complex systems or to improve certain aspects of such systems.

[1] The Deltares Institute was formed in 2008 from two independent institutes and parts from two others. One of these institutes was WL | Delft Hydraulics.
[2] E-mail: arjen.markus@deltares.nl

1. Flood risks in the Netherlands

Most rivers in the Netherlands are surrounded by dikes to prevent the low-lying land from flooding. For the past ten or fifteen years, the safety of these dikes has been the subject of political and technical debate: are the dikes high and strong enough? Should we widen the area between the dikes so that there is more room for the river water, thus lowering the water levels and therefore the risk of flooding? It has also become clear that the traditional approach to assessing the risk, looking at the water levels *in* the river bed and determining whether it does or does not exceed the dikes' crests is insufficient: if a dike breaks, the water will flow onto the surrounding land, thereby lowering the river levels downstream, but increasing the water level at the land side of the dikes elsewhere, undermining their strength, as the water flows into other (smaller) rivers, increasing the local river level. The nett result can be both positive and negative – these effects are collectively known as *effects of river system behaviour on flood risk*. The increase in river levels can be up to 1 m, while the (beneficial) decrease is limited to 10 cm (*cf.* Van Mierlo et al., 2007).

The area considered in this study contains the two major rivers in the Netherlands: the Rhine and the Meuse (figures 1 and 2). Their branches enclose a large, agricultural region and run along two major cities, Arnhem and Nijmegen. The total population is circa two million people. All in all there are 11 *dike rings* (an area protected against floods by dikes and higher grounds).

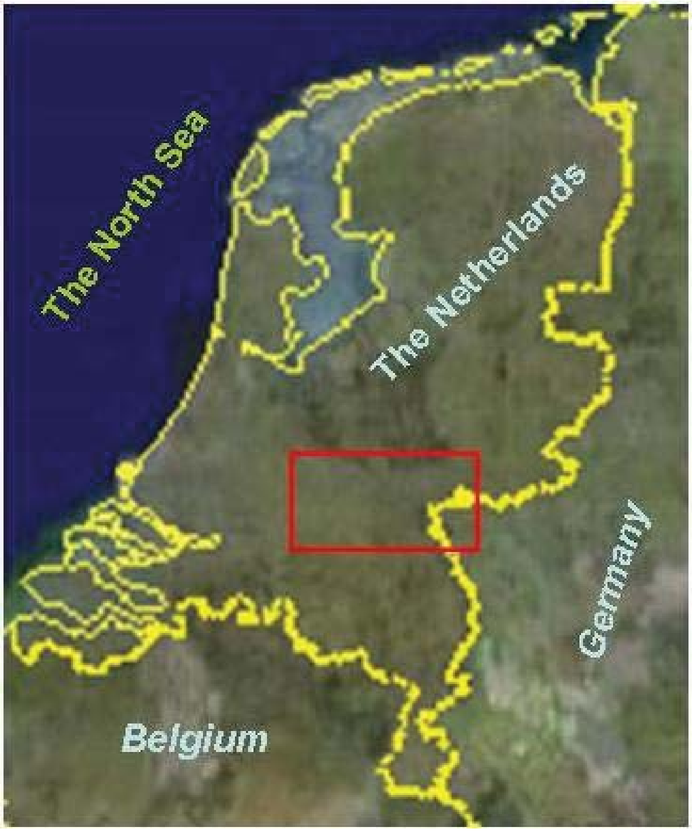

Figure 1. Geography of the area

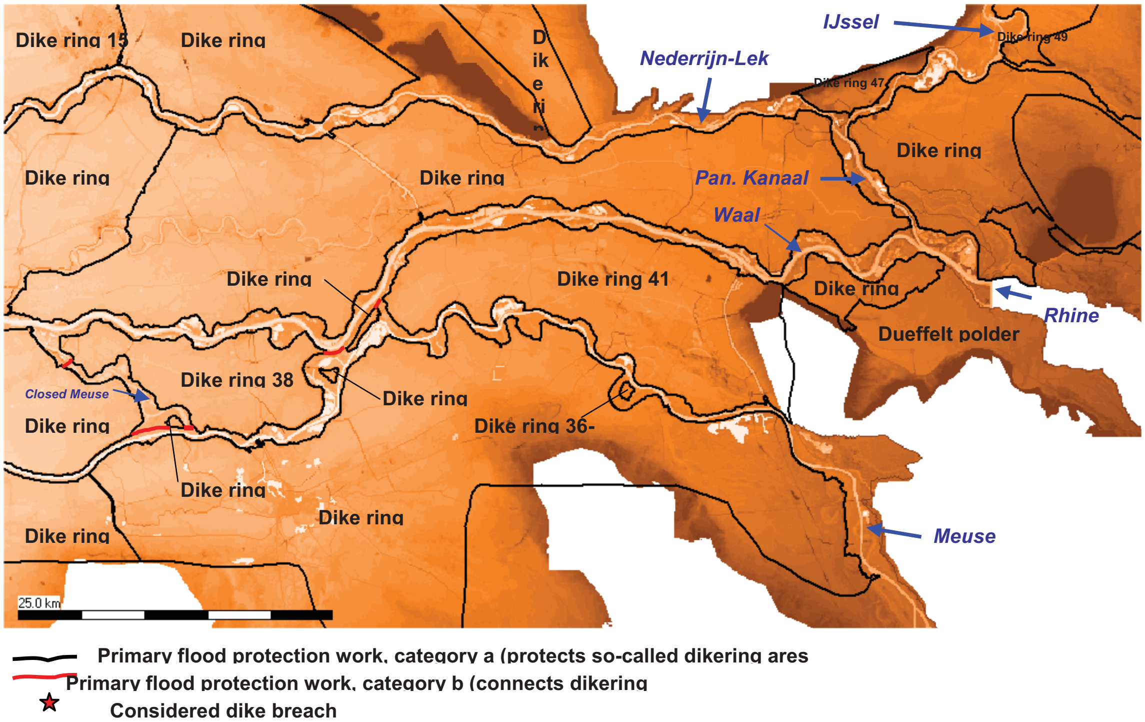

Figure 2. The two major rivers Rhine and Meuse in the study area

The study we present here has the purpose to show:

How effects of river system behaviour can be evaluated in a practical way

How the risk of flooding in a such an area can be estimated, not only the chance that a dike breach occurs, but also the consequences in terms of economic damage and potential loss of lives.

The results of the risk assessment using the computational framework described in this paper have been presented elsewhere (*cf.* Van Mierlo et al. 2008). This paper focuses on the practical aspects of the computations:

Section 2 describes what components can be distinguished in the modelling of the river system and the consequences of a flood.

Section 3 describes the computational framework and its practical implementation using PCs and a Linux cluster.

In section 4 we present a more formal description of the system in terms of a *tuplespace*. The purpose is to highlight the underlying generic structure, and to show how similar systems could be designed or described.

Section 5 contains a few observations about the framework: what lessons did we learn?

Related work is discussed in section 6.

Finally, section 7 summarises the study.

2. Components in the analysis

To analyse the consequences of a dike breach in the study area we set up a system comprising of the following components:

Component 1: Determination of hydraulic loads without considering effects of river system behaviour

Initially, hydrodynamic calculations are carried out for the chosen geographical model, assuming absence of river system behaviour effects. That is, the hydraulic loads on the dikes are computed assuming that the entire flood wave passes through the system without any dike failure. These computations are carried out for a range of peak discharges at the upstream boundary of the system using the SOBEK modelling system. The results are stored in a hydraulic data base.

The numerical and hydrodynamic aspects of this modelling system are described in detail by Dhondia and Stelling (2004). It belongs in fact to a class of hydrodynamic models that emphasizes particular simplifications to the equations, so that a more efficient computation is possible (*cf.* Sulis, 2009).

Component 2: A representative set of Monte Carlo realisations, conditional upon failure

The general expression for a flood risk R for a certain time interval (0,t) is given by:

$$R = E(D) = \int D(\underline{x})\, f(\underline{x})\, d\underline{x} \qquad (1)$$

Where

$\underline{x}$ the vector with all the stochastic parameters
$f(\underline{x})$ is the joint probability distribution function of $\underline{x}$.
$D(\underline{x})$ is the capitalised value of the damage in (0,t)
$E(..)$ is "expected value"

Elements of the vector $\underline{x}$ which play a role in the problem are: the river discharge, the wind speed, the sea level, soil properties, dike lining, emergency measures, hydraulic roughness of inundated areas, behaviour of secondary dams, etc. These quantities are defined for every point in time in (0,t) and for every point in space.

The integral (1) can be evaluated using a Monte Carlo procedure. In such a calculation a set of random variables $\underline{x}$ is generated and the series of events that takes place in the flood area, are determined. This is a complex but fully deterministic analysis. All the water levels, the waves, the dike strengths etc. in the entire area are known for the period under consideration. If the combination $\underline{x}$ leads to an initiation of flooding somewhere in the area, all consequences of this event for the rest of the area can be considered.

At selected potential breach locations reliability analyses per section are carried out using Crude Monte Carlo runs steered by the Prob2B program (*cf. Courage et al.*). The realizations comprise properties of the dikes regarding the considered failure mechanisms as well as peak discharges at the upstream boundary of the geographical model. The load on the analyzed dike sections are interpolated using the hydraulic database from the first component. Loads and resistances are compared using performance functions for selected failure mechanisms.

The results of this step is a representative set of realizations conditional upon failure (at least one dike section fails) and the complementary set of realizations, in which no failure and hence no flood damage occurs. If there is a failure, all data will be stored for the third component. The data consist of:

Discharge time functions of the rivers Rhine and Meuse
Strength properties for all potential breach locations
Breach properties, including possible random quantities
The location of the failure
The failure mechanism

Component 3: Hydrodynamic calculations, allowing for effects of system behaviour

In this part the hydrodynamic consequences (i.e. determination of the flooding pattern) including the effects of dike failures and overflow of dikes are determined for the representative set of realisations obtained from component 2. This was done by means of SOBEK computations.

The entire region is schematised as a two-dimensional square grid with grid cell sizes of 100 m. The computational cycle consists of the following steps:

Propagation of the river water via the one-dimensional network
Evaluation via the so-called RTC module (RTC is the abbreviation of "real-time control") of all the relevant failure mechanisms as well as breach development
Propagation of the river water through the two-dimensional grid when a breach has occurred according to the second step

More specifically, a local dike breach is modelled as a 1D branch, which is connected to 2D grid cells, respectively located at the river side and at the dike ring side of a dike breach location. The 1D branch accommodates a weir, which is lowered and broadened in accordance with the applied Verheij and Van der Knaap (2002) breach growth formula. Hence, dike breaches can only occur at "1D dike breach branches", while "2D river dike grid cells" cannot fail but are overtopped as soon as river levels exceed local crest levels. Details can be found in Van Mierlo et al. (2008).

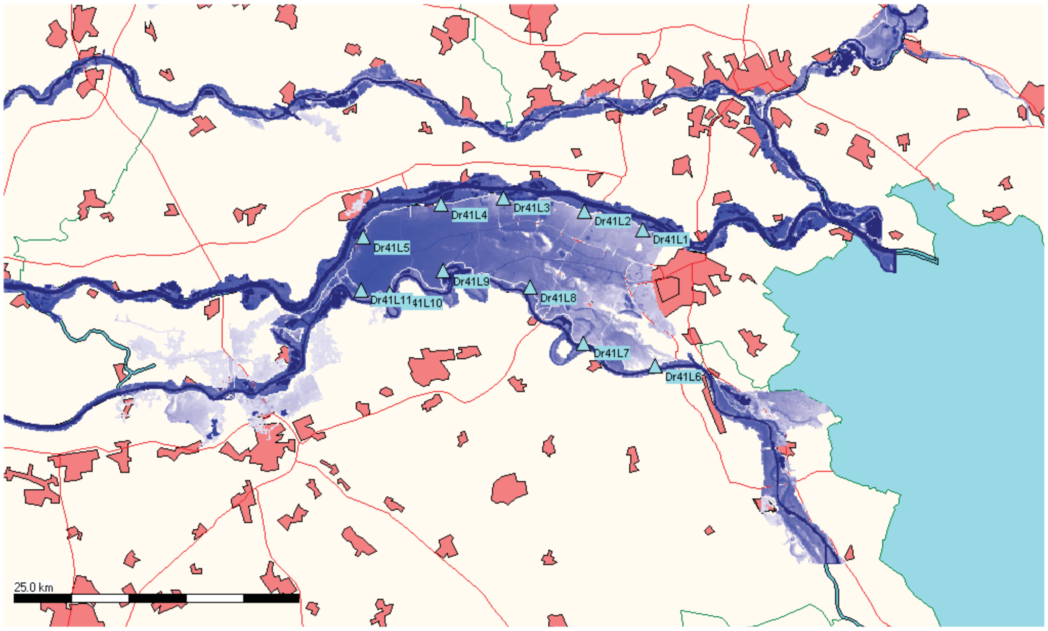

Figure 3. Typical result of the flow computation – various dike rings are flooded (Maximum flow rate for the Rhine is 18,900 m^3/s and for the Meuse 4,300 m^3/s.)

The main output of the hydrodynamic model is the flood pattern of each scenario (*cf.* Figure 3). For each 2D grid cell, SOBEK provides its maximum water depth, its maximum flow velocity and the speed at which water levels rise. This output data is used for determining the flood consequence (i.e. damage and victims) of each scenario.

Component 4: Determination of flood consequence (damage & victims)

Flood consequence is determined for the representative set of realizations conditional upon failure. More precisely, the direct economic damage as well as the expected number of human casualties is computed with the HIS-SSM program using the flooding patterns determined in the previous component as main input (*cf.* Huizinga et al., 2004).

No damage and no victims are assumed for the set of Monte Carlo realizations, in which no dike failure occurred (e.g. the complementary set of realizations).

Component 5: Determination of Flood Risk

Using the results from the previous steps we can determine the risk:

$$R = E(D \mid F)\, P(F) \qquad (2)$$

D is the damage for an arbitrary scenario, P(F) is the system failure probability, which follows from the reliability analysis and *E(D | F)* is the average damage.

3. The computational framework

The various components exist as separate programs on different computers, as they were developed as standalone applications by different groups: PCs in two different locations and a Linux cluster, consisting of some 75 nodes with a total of 150 CPUs. For this project the programs were made to cooperate using ad hoc methods, but little or no adaptation of the programs themselves was necessary *(cf.* Figure 4).

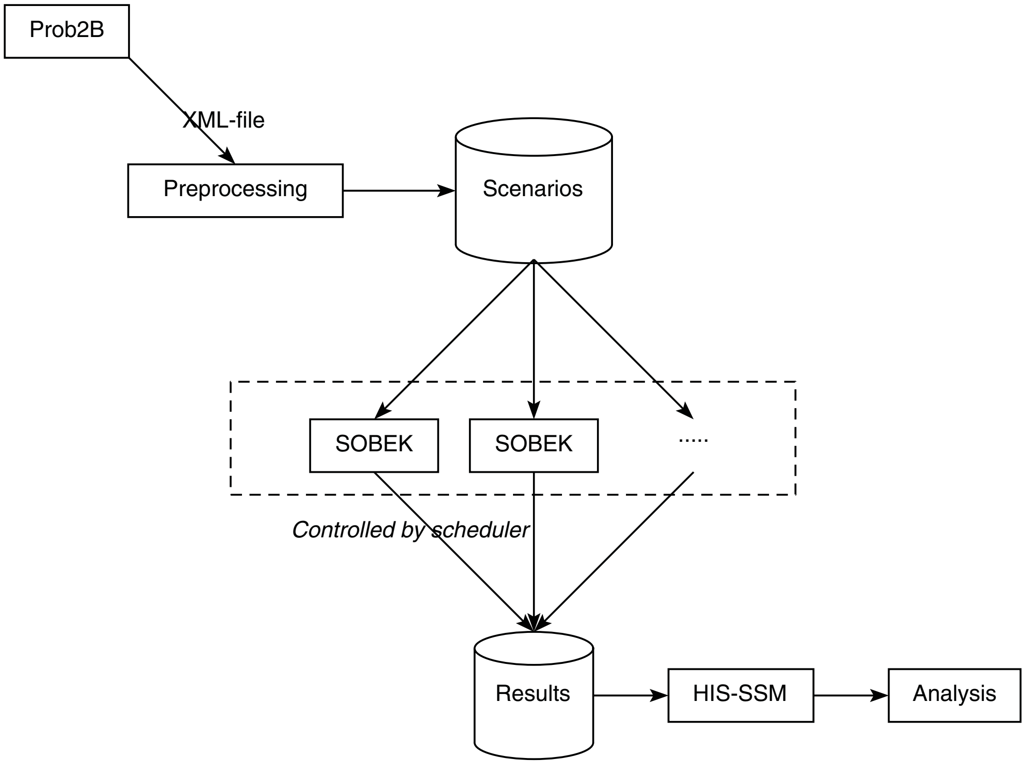

Figure 4. Overview of the programs in the framework

The Monte Carlo program to select the dike strength parameters and the flow conditions was developed for the purposes of this and similar projects, but we chose the XML format as the means to transfer the selected sets to the next step.

XML proved to be an easy-to-use format, as it fitted the structure of the information very well: within each scenario we had data for the floods, characterised by the maximum flow rate per river and the time difference between the occurrences of the maxima and sets of strength parameters per dike breach location. Each parameter is uniquely identified by its XML tag, so that there could be no mistake which is which. The Monte Carlo program was developed at a different location (the TNO offices) than the location where the hydraulic computations were to be done (the Deltares), so an unambiguous format like XML is very attractive.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Set-up of XML files for defining the scenarios to be computed by SOBEK -->
<proboxToSobek>
        <scenario id="dr41run4">
                <boundary id="RijnafvoerLobith" type="1">
                        <maxDischarge>18534.00374491226</maxDischarge>
                        <floodWaveShape>50</floodWaveShape
        <delayPeakMeuse>0</delayPeakMeuse>
                </boundary>
                <boundary id="MaasafvoerVierlingsbeek" type="1">
                        <maxDischarge>3693.736532262028</maxDischarge>
                        <floodWaveShape>50</floodWaveShape>
                        <delayPeakMeuse>0</delayPeakMeuse>
                </boundary>
                <dikeBreach id="Dr41L7">
                        <evaluateBreaching>3</evaluateBreaching>
                        <d_aquifer>14.143837723240722</d_aquifer>
                        <l_piping>70.52933625966438</l_piping>
                        <d_topLayer>1.8176598219403994</d_topLayer>
                        <h_hinterDefault>7.988899434135852</h_hinterDefault>
                        <gamma_sat>18.098665835608376</gamma_sat>
                        …
                        <T0>0.1</T0>
                        <uc>0.2</uc>
                        <v1>0.2</v1>
                        <Tp>3.0</Tp>
                </dikeBreach>
                …
        </scenario>
        <scenario id=”dr41run5>
                …
        </scenario>
</proboxToSobek>
```

Figure 4. Fragment of the XML files used to transfer the scenario information.

The hydraulic computations were done in two steps. First the input was processed on a PC (parts of the hydraulic modelling system have been written in Visual Basic and it was not feasible to convert these for a Linux environment) and then the resulting preprocessed input files were copied to a network disk:

The difference between each scenario consists of the upstream boundary conditions of the rivers Rhine and Meuse, the computational period and the dike strength parameters.

We used a Tcl program (*cf.* Welch et al. 2005) to read the XML file containing the scenario details, prepare the timeseries describing the upstream boundary conditions, fill in the data in the input files and copy everything into a new directory on the network disk.

Using Tcl made these tasks very easy to implement (see also the appendix):

The *clock format* command was of great use to manipulate the date and time strings required for the timeseries.

Via the *file* command and its subcommands it is easy to create and copy directories and files.

The *string map* command was used to fill in the timeseries (once constructed as a string with the proper format) in the input file, as shown in the code fragment below:

```
# main --
#     Select the flood wave in question, construct the timeseries and
#     insert into the boundary.dat file
#

set datafile [lindex $argv 0]
set rate     [lindex $argv 1]
set template [lindex $argv 2]
set key      [lindex $argv 3]
set reftime  [lindex $argv 4]
set delay    [lindex $argv 5]
set expand   [lindex $argv 6]
set output   [lindex $argv 7]

#
# Get the water levels as function of time for each maximum flow rate
#
source [file join [file dirname $argv0] .. templates floodwaves.data.$datafile]

#
# Construct the actual timeseries from that information
```

```
#
set sobekseries [constructTimeseries $reftime $delay $floodwaves($rate,data)]

#
# Fill it in in the file with boundary conditions - replace the right keyword
#
set infile      [open $template]
set contents    [string map [list $key $sobekseries] [read $infile]]
close $infile
set outfile     [open $output w]
puts -nonewline $outfile $contents
close $outfile
```

We took care of reading the XML file using the tDOM extension which renders the XML file as a tree of data (*cf.* J. Löwer, 2000). Writing an XML file is easy, reading it is more complicated. Fortunately there are many tools available for that in almost any programming language.

On the Linux cluster another Tcl program was run automatically every ten minutes via the cron daemon. This program monitored the progress of the computations:

If there was a new directory with the input for a new computation and we don't have too many jobs running already, it schedules the computation. (We wanted to avoid filling up the entire cluster, as it is used by others as well and our computations took several days each). To mark the fact that it is scheduled, a file with the name "scheduled" is created. This has two purposes:

- Mark the directory/computation as being scheduled
- Monitor how long a job is scheduled (by looking at the creation time of the file)

If a computation has started running, the shell script that starts it creates a file "running" – the difference in creation time between this file and the file "scheduled" is the time the job was scheduled but not running. When there are many jobs running, there may not be a free slot for the job, so this was an indication of the performance of the cluster.

When the computation was completed or the computational program stopped for some other reason, the shell script creates a file "done". This allows us to monitor the run-time of the computations.

The Tcl program checks each finished computation: sometimes the computation can become unstable or some other numerical problem occurs and then the hydraulic modelling program may stop before completion. By examining one of the output files from this program, it determines whether the computation was successful or not (to mark this fact, it creates yet another file "analysed").

The one-but-last step in the sequence of programs is to analyse the results of the hydraulic computation to estimate the potential number of casualties and the economic damage. To this end, a third Tcl program is used, running on PC: it examines the directories containing the successfully completed computations, copies various files from the network disk to the local work directory, and runs the HIS-SSM program. Then the directory is marked with yet another file, "his-ssm-done".

Each computational step produces detailed information of which only a part is required in the subsequent step. For the analysis of the collection of computations the results are condensed into:

The potential number of casualties and economic damage per subregion

An overview of the dike breach characteristics:

Did failure occur due to loads from the river or dike ring side?
What failure mechanism was responsible?
Time of the failure
Maximum breach width
Maximum discharge through the dike breach

Most of these steps are highly automated, using dedicated Tcl programs and *cron* tasks on the Linux cluster for scheduling, but the configuration files for the schematisation of a particular region requires human action: The input files contain information on the hydraulic boundary conditions, location of potential breaches, considered failure mechanisms and associated strengh parameters as well as the delineation of subareas for which the flood risk should be determined. The automatic procedures rely on keywords in the modified input files: this modification must be done manually and checking the correctness is essential, as otherwise the computations will be useless.

4. A formal view

As we described in the previous section, the analysis consists of various steps and even a single hydraulic computation goes through a sequence of phases. Each step and each phase is treated by a different program. We can describe the transition from one step to the next by means of a *tuplespace* (*cf.* Gelernter, 1985, Carriero, 1994).

Tuplespaces are a remarkably versatile yet simple client-server concept. They consist of sets of ordered collections of information, called tuples, and three basic operations, viewed from the client:

out: the client puts a new tuple in the server's database.
in: the client selects a tuple from the database that matches a certain pattern. The tuple remains in the database.
read: like the *in* operation, but the tuple is removed from the database.

In our system each individual computational step can be identified by the following *tuple*:

The job-ID
The state of the computation

The job-ID identifies the precise input data and results:
Inputs:
The set of dike strength parameters and flow conditions (different for each scenario)
The input describing the model area (fixed)
Outputs:
The hydraulic results
The estimates of casualties and economic damage

In terms of tuplespaces, each individual computation can be regarded as a tuple of these two elements (the inputs and outputs are not used for the selection) and the various programs read tuples from the database and produce new tuples. The table below describes all the steps involved:

Program/step	*Operation*	*Tuple/pattern*
MC-program produces a scenario	Out	Scenario given, state: *new*

Preparation of hydraulic computation	Read	Select a tuple with pattern “state=new”
Preparation of hydraulic computation	Out	Scenario given, fixed input given, state: *ready*
Scheduling hydraulic computation	Read	Select a tuple with pattern “state=ready”
Scheduling hydraulic computation	Out	Scenario given, fixed input given, state: *scheduled*
Deciding to schedule a new hydraulic computation	In	Select all scenarios with pattern “state=running”
Cluster scheduler	Read	Select a tuple with pattern “state=scheduled”
Cluster scheduler	Out	Scenario given, fixed input given, state: *running*
Hydraulic program started	Read	Select a tuple with pattern “state=scheduled”
Hydraulic program finished	Out	Scenario given, fixed input given, hydraulic results available, state: *done*
Analysing hydraulic result	Read	Select a tuple with pattern “state=done”
Analysing hydraulic result	Out Or: Out	Scenario given, fixed input given, hydraulic results available, state: *successful* Scenario given, fixed input given, no useful hydraulic results, state: *crashed*
Estimating casualties and damage	Read	Select a tuple with pattern “state=successful”
Estimating casualties and damage	Out	Scenario given, fixed input given, hydraulic results available, estimates available, state: complete
MC-program analysing	Read	Select a tuple with pattern “state=complete”

The various computational states a scenario has to go through are more or less sequential, but the programs that are responsible for each of these stages can work in parallel: there is no need to fully compute each scenario from the beginning to the end, before starting another one. But it is important that only one program at a time handles a particular scenario, hence the *read* operations: it takes the scenario out of the server database and only when the program is finished, is it put back, with a different state. By selecting only a tuple whose state matches a particular pattern, we ensure that the computations are performed in the right order.

In a more complex system the state that results in a particular step may vary, guiding the further steps. In our case, there is one somewhat trivial occasion for this: the hydraulic computation could be unsuccessful – the state *crashed* – the flow over a dike breach might induce very large flow velocities, resulting in numerical instabilities, or there was a hardware/network failure so that the computation could not complete.

Such events required that we investigate the reasons for the failure: a hardware failure meant re-scheduling the computation, whereas a numerical problem has to be solved in a different way.

While normally one thinks of a dedicated program to function as a server, the network disk served very well for this purpose. By keeping the files for the scenarios in separate directories we created in fact a simple way to store the tuples and have them available for each client (except for the MC-program, as that did not run on the same network):

- All technical problems of ACID-proof database management systems are solved by the file server responsible for the network disk.
- The state of the computation is represented by the presence of certain files and no more than that. The creation time of these files could in fact be used to monitor the performance of the system.

As the run-time of each hydraulic computation was monitored, we looked into the relationship between the maximum flow rate, which is also a measure of the period that is to be simulated, and the run-time (*cf.* Figure 6). Some computations did not exhibit the expected dike breaches (but the dikes would overflow without breaking) and one question was whether that influenced the run-time. The graph, however, does not reveal any obvious differences between these two groups of computations, showing that the occurrence of a dike breach in the computation has no significant influence on the performance.

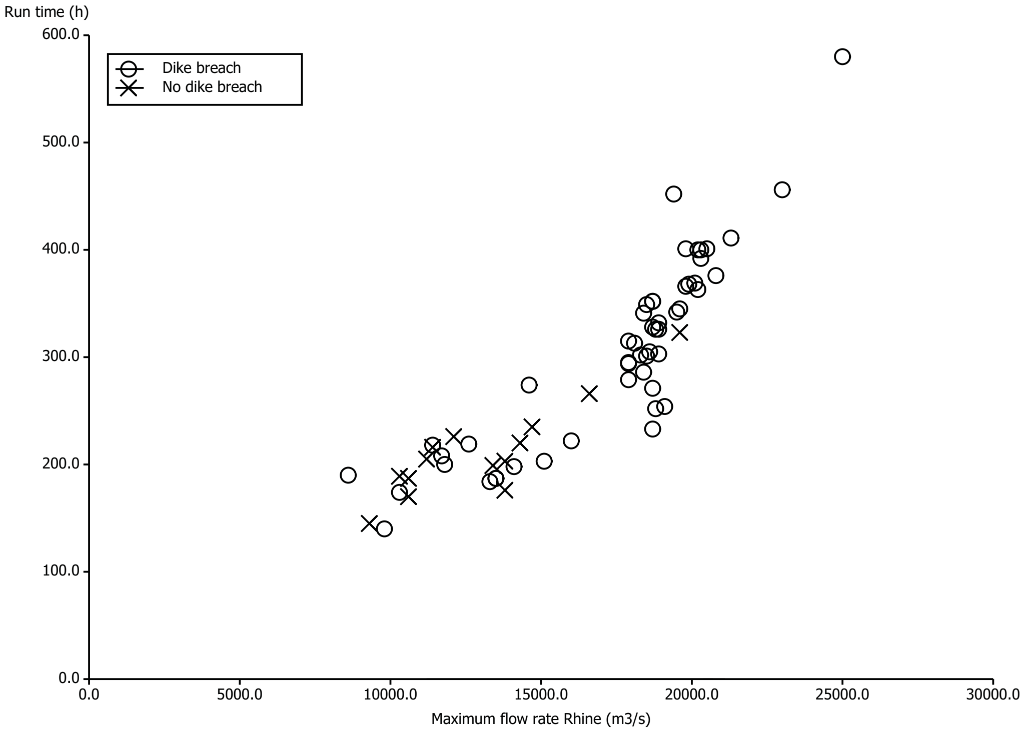

Figure 6. Time required for a computation as function of the maximum flow rate for the Rhine

5. Lessons learned

An automated framework like the one described here requires more “nurturing” than one might anticipate: This is due to:

- the length of the individual computations: a lot can happen in a few days, like a temporary network failure, even though the machines comprising the cluster and the cluster itself are very reliable.
- the number of computations: you simply do not want to check each input file manually, after all, that is one of the things the automation is for.

The automation itself: preparing the input files for the computations should be a smooth process and once automated, it is. Problems arise when you conclude that something has to be changed.

To elaborate the last point: in our case we wanted a series of computations with an alternative, smaller, model area. Because several months had passed, the details of which files were used in which directory were blurred. It was therefore very easy to forget to update one or two of the basic input files, leading to erroneous computations. *The procedure to create the input sets should not only be automatic, it should also be reproducible with different bases of input data.*

While it may seem an open door, it is vital that all programs involved have a “batch mode”: several of the programs we used were developed for interactive use on Windows, and therefore displayed a graphical user-interface, even if the only action the user can take when the program is running is to stop it. Such GUIs need not hinder automation, but they should run in a mode that requires no interaction from the user at all.

As the computations each took several days, being able to monitor the progress was another important aspect:

To be able to estimate the time left
To check if the computation produces reasonable results

Sometimes the computations did not show any dike breachs contrary to our expectations. By examining the intermediate results we were able to identify the cause – slight differences in the geometrical data to select the computational parameters in the Monte Carlo program and the data used in the hydraulic program.

All in all the most important lesson is:

Make sure that you can check exactly what input was used for a computation, that you can check the processing steps and that you can explain the results.

This is nothing new, but with a large number of computations and their respective sets of input and output data, generated with as little human action as possible, it is all the more important.

6. Related work

Controlling large sets of computations has received quite some attention in the literature, for instance in general scientific settings, where very large computations are distributed over multiple computers, colloquially known as GRID computing. But also on a smaller scale, in interactive hydraulic and hydrological modelling, the use of a set of computers for managing a set of computations is being considered. Sulis (2009) presents an interactive system for determining the optimal rules for reservoir management using a GRID environment.

Prodan (2007) describes a generic system called ASKALON where the entire workflow (that is: the set of input data, the various steps in the computation and the set of output data, as well as the relation between these) is analysed and scheduled in such a way that optimal use is made of the available resources.

Paramount to the approach he describes is that the various steps are kept under tight control of the services that schedule the various computational programs. Prodan presents a formal model for this system. This way complex, *hierarchical*, workflows, containing iterations and decisions based on the outcome of computational steps, can be handled. The workflows are first converted into directed acyclic graphs. Then each activity can be scheduled and monitored. If necessary, the converted workflow or parts thereof can be restructured or rescheduled.

To be able to effectively predict the performance the system also uses a training phase during which the execution times of the activities are measured.

When comparing Prodan's system to the *tuplespace* approach presented here, we can see that the current system does not involve iterations and only one decision (whether a particular computation has succeeded or not and what to do when it failed). The workflows are therefore much simpler. This, however, is by no means a limitation of the tuplespace model. Suppose, that the complete computation involves some decision – if a dike is going to break under the given circumstances, we can divert some of the water to another river branch, reducing the pressure on the dike. Of course we do not want to divert too much water, as the dikes along the other branch might then fail. This means the workflow involves a loop with an uncertain number of iterations – we can only estimate the amount of water that should be diverted by trial and error.

The tuplespace model allows us to reschedule this computation:

> After the computation, a separate program examines the results, concludes that some water must be diverted.
>
> The input is adjusted to reflect this diversion
>
> The file "done" is deleted from the working directory, so that the scheduling script will reschedule the computation.

This process can be repeated indefinitely, with no particular complications on the scheduling script. In a similar way other decisions can be arranged for.

An entirely different aspect of managing these computations is the subject of an article by Balis et al. (2008). They present a formal framework for monitoring the activities in a workflow that are being run on geographically distributed computers. They contrast this situation with the situation of running programs on a cluster of computers in a single location. The purpose of their GEMINI monitoring system is to gather useful information about the progress of the computations and to use that for improving the performance. The *practical* problems of gathering this information in the loosely coupled systems they target are much larger than in the computing cluster that we used. The use of locally created files to pinpoint important events in the workflow seems an adequate and simple alternative, though in this study it was set up ad hoc. Nevertheless, the possibility to monitor the progress of these computations is important in any computing environment.

7. Conclusion

The framework we described in this paper, is a collection of pre-existing modelling systems and auxiliary programs developed for just this framework. Combining these tools led to a practical system by which we have been able to assess the risk of flooding in a part of the Netherlands, but it can be applied to other river basins as well. The fact that it is a loose collection of programs, where communication occurs via files on disk and prescribed directory structures, makes it relatively easy to set up and it makes it relatively easy to check what is happening. Using external files for communication between programs often means an impact on performance, but that was no issue in this case.[3]

This is by far the greatest lesson to be learned: the components of such a system must readily allow the input and output to be inspected. If somewhere in the system, either the results or the input is obscure, it is all too easy for errors to propagate. It is also paramount that the steps in the computation are repeatable, because at any point you may make a mistake, no matter how careful you are or one of the computers you use will develop a disk or network problem.

That is also the advantage of the use of pre-existing components that you can trust and regarding the system as a tuplespace: individual tasks with persistent data files that describe them, so that you can restart the whole computation or a part of it when the need arises.

References

Balis, B., Bubak, M., Łabno, B. (2008)
Monitoring of Grid scientific workflows,
Scientific Programming, Vol 16 (2008), pp 205-216

Carriero, N.J., Gelernter D., Mattson T.G, Sherman A.H. (1994)
The Linda alternative to message-passing systems
Parallel Computing, Vol 20 (1994), pp. 633-655

Courage, W.M.G., et al, 2007
Prob2BTM: Installation and Getting Started,
manual. TNO report 2007-D-R0887, TNO Built Environment and Geosciences
See also: Prob2B on www.tno.nl

Dhondia, J.F. and Stelling, G.S., 2004,
Sobek 1D-2D integrated hydraulic model for flood simulation – its capabilities and features explained,
in sixth International Conference on Hydroinformatics, Singapore,
Liong Phoon & Babovic (eds),
World Scientific Publishing Compagny, ISBN 981-238-787-0.

Gelernter, D., 1985
Generative communication in Linda,
ACM Transactions on Programming Languages and Systems, volume 7, number 1, January 1985
See also: http://en.wikipedia.org/wiki/Tuple_space for more information

Huizinga, H.J. et al., 2004,
HIS-Schade en Slachtoffermodule (in Dutch),
version 2.1, manual. DWW-2005-004. RWS Dienst Wegen Waterbouwkunde.

Hunter, N.M., Bates, P.D., Hewitt, M.S. and Wilson M.D. (2007)
Simple spatially-distributed models for predicting flood inundation: A review,

[3] If the issue does arise, other techniques are available, such as MPI, but it makes tracing errors more difficult.

Geomorphology, Vol 90 (2007), pp 208-225

Löwer, J., 2000
tDOM – A fast XML/DOM/XPath package for Tcl written in C,
Presented at the First European Tclers' Meeting, Hamburg, 2000
http://www.tdom.org/documents/tDOM3.pdf

Van Mierlo, M.C.L.M. et al, 2007,
Assessment of floods risk accounting for River System Behaviour,
Intl J. River Basin Management Vol 5, No 2 (2007), pp 93-104.

Van Mierlo, M.C.L.M. Schweckendiek, T. and Courage W.M.G., 2008,
Importance of River System Behaviour in Assessing Flood Risk,
Flood Risk Management: Research and Practice – Samuels et al. (eds),
2009, Taylor & Francis Group, London, ISBN 978-0-415-48507-4

Prodan, R. (2007)
Specification and runtime workflow support in the ASKALON Grid Environment,
Scientific Programming, Vol 15 (2007), pp. 193-211

Sulis, A. (2009)
GRID coputing approach for multireservoir operating rules with uncertainty,
Environmental Modelling & Software, Vol 24 (2009), pp 859-864

Verheij, H.J. and Van der Knaap, F.C.M., 2002,
Modification Breach Growth Model in HIS-OM,
WL|Delft Hydraulics, Q3299 (in Dutch).

Welch,B., K. Jones and J. Hobbs (2003)
Practical Programming in Tcl and Tk,
Prentice Hall PTR, 4th Edition
ISBN: 0-13-038560-3, June, 2003

Appendix

This appendix demonstrates some details in the framework: the editing of the input files and the scheduling of the various jobs.

Preparing the input files

The boundary conditions for typical flood waves are available for each maximum flow rate at steps of 100 m^3/s for a wide range of flow rates. In the program the appropriate series is selected based on the maximum flow rates for Rhine and Meuse from the Monte Carlo simulation in component 2. This flow rate is specified as a list of times (in days since the start of the wave) and the flow rate at that moment. The following simplified procedure produces the timeseries in the format that is expected:

```
proc constructTimeseries {begin series} {
    set sobekseries {}
    set begintime [clock scan $begin]
    set offset    [lindex $series 0]
    foreach {time rate} $series {
        set seconds [expr {int(86400*($time-$offset))}]
        set datetime [expr {$begintime+$seconds}]
        set sobektime [clock format $datetime -format "'%Y/%m/%d;%H:%M:%S'"]
        lappend sobekseries "$sobektime $rate <"
    }

    # Trick: using a list suppresses an end-of-line at the end!
    return [join $sobekseries \n]
}
```

The result is then inserted into the input file using the [string map] command. The variable key has the value “@RIJN@” or “@MAAS@”. These strings are found in the template file and are replaced by the timeseries.

```
    #
    # Construct the actual timeseries
    #
    set sobekseries [constructTimeseries $reftime $delay $floodwaves($rate,data)]

    #
    # Fill it in in the file with boundary conditions – replace the right keyword
    #
    set infile      [open $template]
    set contents    [string map [list $key $sobekseries] [read $infile]]
    close $infile
    set outfile     [open $output w]
    puts -nonewline $outfile $contents
    close $outfile
```

The result:

Template:

```
D2LI id 'l_RijnafvoerLobith' ty 1 q_ dt 1 0 0
TBLE
@RIJN@
tble    d2li
```

After replacing the keyword “@RIJN@”:

```
D2LI id 'l_RijnafvoerLobith' ty 1 q_ dt 1 0 0
TBLE
'2000/01/01;00:00:00' 1000 <
'2000/01/01;07:02:21' 1080 <
'2000/01/01;17:09:10' 1330 <
…
'2000/01/12;12:56:00' 7080 <
'2000/01/13;01:09:24' 7440 <
'2000/01/13;13:06:31' 7800 <
'2000/01/14;01:29:34' 8160 <
'2000/01/14;22:48:08' 8660 <
'2000/01/15;11:20:15' 8900 <
'2000/01/16;06:00:25' 8660 <
'2000/01/17;03:08:03' 8160 <
'2000/01/17;14:27:10' 7800 <
'2000/01/18;04:39:12' 7440 <
'2000/01/18;19:35:54' 7080 <
'2000/01/19;08:53:13' 6830 <
….
'2000/02/02;23:32:29' 1580 <
'2000/02/03;18:41:02' 1330 <
'2000/02/04;09:51:24' 1080 <
'2000/02/04;20:25:00' 1000 <
tble    d2li
D2LI id 'l_7' ty 1 q_ dw 1 0 0
```

Scheduling the jobs

The scheduling program scans the directory (using the [glob] command) holding all subdirectories containing the scenarios for computations that have not run yet and have not been scheduled yet. The criterium is fairly simple: there should not be a file that indicates the job is scheduled, is currently running or has already been done.

If such a subdirectory is found and there are not too many computations running already, it is added to the cluster's work queue. If on the other hand a subdirectory is found with a file "done", the program checks if it has already been analysed. The analysis is limited to the check whether it has finished normally.

```
# main --
#     Loop over the subdirectories and see if there is a subdirectory
#     holding an interesting job
#
set env(SGE_CELL)         …
… and other environment variables …

foreach d [glob -nocomplain -type d *] {
    if { [file exists [file join $d CMTWORK]] } {
        if { ! [file exists [file join $d done]] } {
            if { ! [file exists [file join $d running]] } {
                if { ! [file exists [file join $d scheduled]] } {
                    if { [canSchedule] } {
                        scheduleJob $d
                    }
                }
            }
        } elseif { ! [file exists [file join $d analysed]] } {
            analyseJob $d
        }
    }
}
```